



**C O N F E R E N C E  
P R O C E E D I N G S**

**USENIX Systems Administration  
(LISA VIII) Conference**

**September 19-23, 1994  
San Diego, California**

For additional copies of these proceedings contact

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Telephone: 510-528-8649

The price is \$22 for members and \$29 for nonmembers.

Outside the U.S.A. and Canada, please add  
\$10 per copy for postage (via air printed matter).

Past USENIX Large Installation Systems Administration Workshop  
and Conference Proceedings (price: member/nonmember)

Large Installation Systems Admin. I Workshop	1987	Philadelphia, PA	\$4/\$4
Large Installation Systems Admin. II Workshop	1988	Monterey, CA	\$8/\$8
Large Installation Systems Admin. III Workshop	1989	Austin, TX	\$13/\$13
Large Installation Systems Admin. IV Conference	1990	Colorado Spgs, CO	\$15/\$18
Large Installation Systems Admin. V Conference	1991	San Diego, CA	\$20/\$23
Large Installation Systems Admin. VI Conference	1992	Long Beach, CA	\$23/\$30
Large Installation Systems Admin. VII Conference	1993	Monterrey, CA	\$25/\$33

Outside the U.S.A. and Canada, please add \$8  
per copy for postage (via air printed matter).


Copyright 1994 by The USENIX Association.  
All rights reserved.

This volume is published as a collective work.

ISBN 1-880446-64-2

AFS and Transarc are registered trademarks of Transarc Corporation.  
Alantec and PowerHub are trademarks of Alantec.  
AppleTalk is trademark of Apple Corp.  
System V is a trademark of Unix System Laboratories.  
Cisco and AGS+ are trademarks of Cisco Systems, Inc.  
DEC, VAX, VMS, and Gigaswitch are trademarks of Digital Equipment Corp.  
EtherSwitch is a trademark of Kalpana.  
Ethermeter is a trademark of NAT.  
Ethernet is a trademark of Xerox.  
Multinet is a trademark of TGV.  
SAS is a trademark of SAS, Inc.  
SQL\*Plus and Oracle are trademarks of the Oracle Corp  
Sniffer is a trademark of Network General  
Sun, SunOS, and SPARC are trademarks of Sun Microsystems.  
Unix is a registered trademark of Unix System Laboratories.  
VM, AIX, MicroChannel, and RS/6000 are trademarks of IBM Corp.

USENIX acknowledges all trademarks appearing herein.

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste. 

**USENIX Association**

**Proceedings of the Eighth  
Systems Administration Conference  
(LISA VIII)**

**September 19-23, 1994  
San Diego, CA, USA**



# TABLE OF CONTENTS

Acknowledgments .....	iv
Preface .....	v
Author Index .....	vi

## Plenary Session

**Wednesday (9:00-10:30)**

**Chair: Dinah McNutt**

Opening Remarks and Announcements

*Dinah McNutt, Zilker Internet Park, Inc.*

Keynote Address: The Road To UNIX – A Report from the Fourth Estate

*Jack Stanley, Houston Chronicle*

## System Configuration

**Wednesday (11:00-12:30)**

**Mark Verber**

Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin .....	1
<i>Dr. Magnus Harlander, GeNUA mbH, München, Germany</i>	
Config: A Mechanism for Installing and Tracking System Configurations .....	9
<i>John P. Rouillard &amp; Richard B. Martin, University of Massachusetts at Boston</i>	
Towards a High-Level Machine Configuration System .....	19
<i>Paul Anderson, University of Edinburgh</i>	

## Automation

**Wednesday (2:00-3:30)**

**Hal Stern**

OMNICONF – Making OS Upgrades and Disk Crash Recovery Easier .....	27
<i>Imazu Hideyo, Matsushita Electric</i>	
Automated Upgrades in a Lab Environment .....	33
<i>Paul Riddle, University of Maryland, Baltimore County</i>	
Tenwen: The Re-engineering Of A Computing Environment .....	37
<i>Rémy Evard, Northeastern University</i>	

## The Toolbox

**Wednesday (4:00-5:00)**

**Pat Parseghian**

Kernel Mucking in Top .....	47
<i>William LeFebvre, Argonne National Laboratory</i>	
Handling Passwords with Security and Reliability in Background Processes .....	57
<i>Don Libes, National Institute of Standards and Technology</i>	

## Plenary Session

**Thursday (9:00-10:30)**

**Chair: Dinah McNutt**

Keynote Encore: Breaking into Banks – Security Lessons Learned from Financial Services  
*Dan Geer, OpenVision, Inc.*

## Software Configuration

**Thursday (11:00-12:30)**

**Paul Evans**

Soft: A Software Environment Abstraction Mechanism .....	65
<i>Rémy Evard and Robert Leslie, Northeastern University</i>	
Beam: A Tool for Flexible Software Update .....	75
<i>Thomas Eirich, University of Erlangen-Nürnberg, Germany</i>	
Depot-Lite: A Mechanism for Managing Software .....	83
<i>John P. Rouillard and Richard B. Martin, University of Massachusetts at Boston</i>	

## Automation, the Sequel

**Thursday (4:00-5:30)**

**Neil Todd**

SENDS: a Tool for Managing Domain Naming and Electronic Mail in a Large Organization .....	93
<i>Jerry Scharf, Sony Electronics; Paul Vixie, Vixie Enterprises</i>	
Getting More Work Out Of Work Tracking Systems .....	105
<i>Elizabeth D. Zwicky, Silicon Graphics</i>	
Managing the Ever-Growing To Do List .....	111
<i>Rémy Evard, Northeastern University</i>	

## The User Environment

**Friday (9:00-10:30)**

**Trent Hein**

Speeding Up UNIX Login by Caching the Initial Environment .....	117
<i>Carl Hauser, Xerox Palo Alto Research Center</i>	
The BNR Standard Login (A Login Configuration Manager) .....	125
<i>Christopher Rath, Bell-Northern Research Ltd.</i>	
Exporting Home Directories on Demand to PCs .....	139
<i>David Clear and Alan Ibbetson, University of Kent at Canterbury, UK; Peter Collinson, Hillside Systems</i>	

## Peek a Boo – I Can See You

**Friday (11:00-12:30)**

**William LeFebvre**

Monitoring Usage of Workstations with a Relational Database .....	149
<i>Jon Finke, Rensselaer Polytechnic Institute</i>	
Adventures in the Evolution of a High-Bandwidth Network for Central Servers .....	159
<i>Karl L. Swartz, Les Cottrell, and Marty Dart, Stanford Linear Accelerator Center</i>	
Pong: A Flexible Network Services Monitoring System .....	167
<i>Helen E. Harrison, Mike C. Mitchell, and Michael E. Shaddock, SAS Institute, Inc.</i>	

## The Automation Revolution

**Friday (2:00-3:30)**

**Tom Christiansen**

Automating Printing Configuration .....	175
<i>Jon Finke, Rensselaer Polytechnic Institute</i>	
Highly Automated Low Personnel System Administration in a Wall Street Environment .....	185
<i>Harry Kaplan, Sanwa Financial Products Co., L.P.</i>	
The Group Administration Shell and the GASH Network Computing Environment .....	191
<i>Jonathan Abbey, The University of Texas at Austin</i>	

# ACKNOWLEDGMENTS

## GENERAL CHAIR

Dinah McNutt, *Zilker Internet Park, Inc.*

## PROGRAM COMMITTEE

Tom Christiansen, *Consultant*  
Trent Hein, *XOR Network Engineering*  
William (Bill) LeFebvre, *Argonne National Laboratory*  
Pat Parseghian, *AT&T Bell Laboratories*  
Hal Stern, *Sun Microsystems, Inc.*  
Jeff Tate, *Wells Fargo Institutional Trust Company*  
Neil Todd, *Swiss Bank Corporation*  
Mark Verber, *Xerox PARC*

## Invited Talks Coordinator

Pat Parseghian, *AT&T Bell Laboratories*

## Work-In-Progress Coordinator

Bryan MacDonald, *SRI International*

## Guru Is IN Coordinator

Paul Evans, *Synopsys, Inc.*

## TUTORIAL COORDINATOR

Daniel V. Klein, *USENIX Association*

## TERMINAL ROOM COORDINATOR

Barb Dijker, *Consultant*

## PROCEEDINGS PRODUCTION

Rob Kolstad, *BSDI*  
Carolyn S. Carr, *USENIX Association*  
Malloy Lithographing, *Inc.*

## USENIX MEETING PLANNER

Judith F. Desharnais, *USENIX Association*

## USENIX EXECUTIVE DIRECTOR

Ellie Young, *USENIX Association*

## USENIX SUPPORT STAFF

Diane DeMartini, *USENIX Association*  
Toni Veglia, *USENIX Association*

## USENIX Marketing Director

Zanna Knight, *USENIX Association*

## VENDOR DISPLAY COORDINATOR

Peter Mui, *USENIX Association*

# PREFACE

LISA '94 promises to be an exciting conference with two high quality technical tracks: one for refereed papers and the second for invited talks. My goals for this conference were to make it difficult deciding which session to attend and to share my philosophy of system administration by focusing on tools and automation.

The key to a successful conference is the program committee. This year's committee took seriously their challenge to produce a high quality technical program. I appreciate the time and care they put into refereeing the submissions.

The biggest thanks goes to those who took the time to submit an abstract. We could not accept all the papers we received, but those who had their papers accepted should be very proud to pass the scrutiny of the program committee. The authors who were not accepted this year should submit again next year with the advantage of having constructive feedback on their submissions. They also make the program committee's job more difficult this year with the quality of their submissions.

No conference is complete without the informal Birds of a Feather sessions and I would like to thank Lee Damon for his efforts. We had BOFs scheduled months in advance of the conference because of his efforts. Paul Evans has lined up some interesting gurus for the Guru is In sessions. As if that is not enough, Bryan MacDonald is coordinating the Work In Progress session. It is probably not too late to sign up for a time slot.

Pat Parseghnian not only did an outstanding job as a program committee member, but she also organized the invited talks. I think you will agree with me that she did a superb job.

I am sure I have left someone out. Naturally, the USENIX office makes my job much easier by being well organized and experienced at putting on a conference of this size. All the planning and organizing through this past year has been accomplished smoothly and professionally. I thank all the staff for their efforts.

Please accept my welcome to the 1994 LISA conference. I hope you will meet your goals in attending and walk away with at least one or two new ideas.

Thanks for coming!

Dinah McNutt

## AUTHOR INDEX

Jonathan Abbey .....	191
Paul Anderson .....	19
David Clear .....	139
Peter Collinson .....	139
Les Cottrell .....	159
Marty Dart .....	159
Thomas Eirich .....	75
Rémy Evard .....	37
Rémy Evard .....	65
Rémy Evard .....	111
Jon Finke .....	149
Jon Finke .....	175
Dr. Magnus Harlander .....	1
Helen E. Harrison .....	167
Carl Hauser .....	117
Imazu Hideyo .....	27
Alan Ibbetson .....	139
Harry Kaplan .....	185
William LeFebvre .....	47
Robert Leslie .....	65
Don Libes .....	57
Richard B. Martin .....	9
Richard B. Martin .....	83
Mike C. Mitchell .....	167
Christopher Rath .....	125
Paul Riddle .....	33
John P. Rouillard .....	9
John P. Rouillard .....	83
Jerry Scharf .....	93
Michael E. Shaddock .....	167
Karl L. Swartz .....	159
Paul Vixie .....	93
Elizabeth D. Zwicky .....	105

# Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin

*Dr. Magnus Harlander – GeNUA mbH, München, Germany*

## ABSTRACT

GeNUAdmin is an automatic system administration tool to control the configuration and administration of Unix workstations of different vendors from a central management host.

All relevant data about the system are kept in a central data repository. The configuration for each machine is extracted from these databases. Configuration files and installation methods are generated for many different architectures.

Before calculating and updating the configuration of each machine many consistency checks using pattern matching or inherent data correlations are performed. This assures data consistency as well as configuration consistency for a large number of machines and users.

The system is written in perl and creates Bourne shell scripts to accomplish its tasks. The prerequisites for using it are minimal.

GeNUAdmin is available without charge.

## Motivation

Wouldn't it be nice to have an *OBSAT* – the “One Button System Administration Tool”? Push the button and it does what you want. However, a cluster of Unix workstations is not a triviality, especially since nature has supplied us with a wide variety of operating systems, different command line options and file formats. So the real world does not allow us an *OBSAT*!

On the other hand, it is possible to represent all relevant data of a system configuration in an abstract notation. Using the knowledge about different file formats and installation methods it should be possible to deal with the maintenance of this abstract data repository and let the real work be done by an automatic system.

By modifying one database it should be possible to add a new user, printer, disk, or modify your existing configuration.

## Introduction

GeNUAdmin's main goal is to automate as many system administration tasks as possible. It enables the system administrator to manage large numbers (hundreds) of machines and users (thousands) from a central server without the need to log in to any of these machines.

By automating many tasks we get the chance to perform consistency checking at many different levels of the configuration process. Using perl [1] as the programming language allows us to perform many powerful and efficient tests at different levels of the configuration process. Using perl's powerful

pattern matching methods and associative arrays, we can check, e.g., for correct format of variable values or for the uniqueness of variables (e.g., for IP addresses or user names).

Consistency checks are performed

- when creating or modifying the databases,
- when parsing the databases,
- when creating the configuration and
- when installing the configuration on the target host.

GeNUAdmin is a tool for automating many tasks but it should not be used by inexperienced system administrators. It is essential to understand the structure and internals of a Unix cluster in order to avoid catastrophic errors.

GeNUAdmin is executed in several steps.

- Parsing the databases
- Creating a new configuration
- Installing the new configuration

Each run level can be executed by hand. So you can run GeNUAdmin without touching the existing configuration. The system administrator has the opportunity to check that the new configuration really looks as expected.

The installation process of a machine's configuration is split into so-called targets. A target might be, e.g., the configuration of */etc/fstab* or the setup of a name server. It is possible to address specific targets on specific hosts for installation. This enables us to test the effect of a modification to the databases on a single machine before reconfiguring all machines at once.

## The Databases

All information relevant to the configuration of GeNUAdmin targets is collected in a few ascii database files on the central server.

### The Structure of the Databases

Some databases (e.g., the `inetgroup.db` and `group.db` files) are laid out in a rather simple tabular format since the related information and information structure is more or less static. Figure 1 shows an example for `netgroup.db`.

```
#####
# netgroup.db: defines all netgroups
#####
host:all-hosts:sun4-hosts pmax-hosts:
host:sun4-hosts::
host:pmax-hosts::
user:all-users:cs-users ee-users:
user:cs-users::
user:ee-users::
```

**Figure 1:** A simple table to define all available netgroups. This table contains mostly redundant information. We use the entries mainly for consistency checks with respect to user and host netgroups from the other databases.

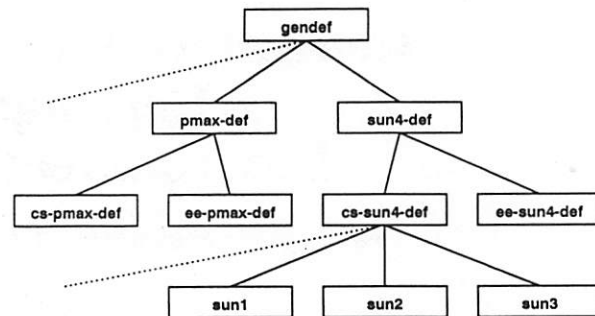
Entities like users, hosts, printers or filesystems however are treated as complex objects with many properties. The objects are stored in a stanza format, which avoids a static information structure and allows us to add properties on demand. All stanza keys however are cross checked with a static list of known keys to avoid typos and inconsistencies in the database itself. Figure 2 shows the format of a stanza entry for the `hosts.db` database.

```
BEGIN_HOST
name      = EntryName
key       = Value
key       = Value
....
default   = DefaultName
END_HOST
```

**Figure 2:** Complex data structures are stored in stanza format. We use it to describe hosts, users, printers and filesystems.

For all objects, it is possible to inherit properties from default entries which can again inherit information from other default entries. The levels of inheritance are unlimited. Figure 3 illustrates a useful constellation of default entries.

It is clear that with this method managing many similar hosts becomes very easy. The entry for a single host becomes very short containing only key/value pairs for the host name, IP address and ethernet address. Figure 4 shows an example for a server entry and two cloned clients.



**Figure 3:** Example for the hierarchical structure of the object oriented databases. `sun4` and `pmax` defaults inherit information from the most generic default `gendef`. Department defaults like `cs-sun4-def` extend this information which finally is used by the hosts `sun1`, `sun2`, ..., etc.

```
BEGIN_HOST
name      = server1
address   = 135.33.44.1 135.33.45.1
cname     = loghost mailhost
ypmaster  = YoUR.NiS.dOMain
....
default   = some-default
END_HOST

BEGIN_HOST
name      = client1
address   = 135.33.44.2
default   = some-default
END_HOST

BEGIN_HOST
name      = client2
address   = 135.33.44.3
default   = some-default
END_HOST

BEGIN_DEFAULT
name      = some-default
ostype    = bsd
....
default   = basic-default
END_DEFAULT

BEGIN_DEFAULT
name      = basic-default
domain    = your.ns.domain.com
....
END_DEFAULT
```

**Figure 4:** Example for entries in `hosts.db`. Server entries may be longish while client entries tend to be minimal.

### List of Databases

The information about the different types of objects is split into several object specific files. The following listing gives an overview over the most important databases and related variables. The `defines.db` file contains a list of variables for

controlling the general behavior of GeNUAdmin. Next to things like path names and mail addresses this file contains also switches for enabling or disabling the execution of certain targets.

The **hosts.db** file contains all data relevant to a host. These are:

- host name, domain name and a list of additional domain names,
- IP and ethernet addresses, subnet masks,
- name server information like CNAMEs and MX records,
- architecture, operating system type and version,
- a list of local filesystems,
- amd related variables,
- a list of hosts for root's `.rhosts` -file,
- a list of host netgroups the host should belong to,
- lists of domains for configuring a primary or secondary name server,
- flags for configuring master or slave yp servers,
- variables for a set of files to keep up-to-date:
  - crontab
  - services
  - inetd.conf
  - syslog.conf
  - resolv.conf
- a flag whether this host is to be managed by GeNUAdmin.

The **disks.db** file contains all relevant data for network filesystems. The data are used for exporting and mounting filesystems via `fstab` or `amd`. The most important filesystem variables are:

- the location of the disk. For replicated filesystems this might be a list of file servers. GeNUAdmin will determine the closest – in terms of network metric – for each host,
- the mount point,
- the original directory name on the file server,
- the mount options,
- an access list based on hosts and netgroups,
- architecture and os-type dependencies, since binaries for a SUN should not be mounted on a HP,
- the preferred mount method: `amd` or `fstab`.

The **user.db** file contains all users for the cluster with the following user properties:

- first name,
- last name,
- user name,
- user id,
- a list of group ids (the first will be the primary group),
- a list of netgroups the user should belong to,
- home directory,
- login shell,
- gecost field data,
- quotas for a list of filesystems,

- expire date for the account. GeNUAdmin will disable the users login shell after this date but the user will get several warnings before getting disabled.

In the next section GeNUAdmin's concept of user management will be explained in detail.

The **printer.db** file contains all available printers.

The **links.db** file is for maintaining a list of soft links on all systems. The entries in this database have tags that may be used to decide which links should be made on which hosts. The decision may depend on the architecture, operating system or a match against a list of host names.

The **net.db** file contains a description of the net topology. With this information about nets, subnets and gateways it is possible to create static routing tables and determine the distance in terms of gateway hops between two hosts. When creating `fstab` entries for multi homed hosts or replicated filesystems, this information is used in order to find the closest file server.

### User Management

All databases are modified using a regular editor, however, a tk/tcl [2] based graphical interface is under development.

The user database `user.db`, however, can be accessed and modified by an home brew RPC server (`usermgrd`). For fast access the data used by `usermgrd` are stored in a dbm file. Therefore the data from the dbm file must be converted to an ascii file in stanza format before using it with GeNUAdmin. Actually GeNUAdmin does this for you.

User management – adding, modifying and deleting users – can all be done by simply modifying the `user.db` database. The modification is performed by communicating with the `usermgrd` through one of two RPC based client programs: `usermgr` and `rpasswd`.

#### usermgr

`Usermgr` is intended for user management by junior (and senior) system administrators.

With optionally restricted access to a subset of users they have the possibility to

- add,
  - delete,
  - enable or disable,
  - modify, and
  - set the password
- of their users.

The access rights are based on

- the location of the home directory,
- group and netgroup membership,
- the maximum quota value,
- the maximum lifetime of the account, and
- a range of user ids.

*Usermgrd* reads the file *auth.db* to learn about the access rights.

To grant free access for senior system administrators wild cards ("\*") can be used in some or all of the fields. Figure 5 shows an example of a small authentication database.

```
#####
# auth.db: granting access to user.db
#####
# user:dir:groups:ngs:quotas:exp:uids
#####
bigmac:/:*:0:0:0-32000
ee:/home:ee:50000:30000:0-10,50-199
cs:/home/cs:cs:20000:30000:200-299
```

**Figure 5:** An example for the authentication database *auth.db*. Three users allowed to modify the database. Bigmac has unrestricted access while access for the users *ee* and *cs* is restricted to users of their department only.

### **rpaswd**

The second program, *rpaswd*, is intended for use by all users and allows them to change their own password or login shell by modifying the central password file.

The server – *usermgrd* – can be told which password file to change and which command to execute after the update of the password file. Therefore one can use NIS master files, local password files and even shadow password files.

If you don't like NIS, this actually might be the method to maintain a central password file. You could *rdist* this file once in a while to all your machines or use *GeNUAdmin* to update the local password files (See the section "Local password files").

### **Consistency**

The client asks the operator for a couple of user attributes and checks with the server whether the attributes are acceptable. It asks the server also for a unique user id. The server checks user name, user id, first and last name for uniqueness. It also checks the format of all known keys (e.g., uid must be numeric). If first and last name already exist, you might add initials of additional names with a period (e.g., John Kennedy -> John.F Kennedy).

After completing the form, a final check for consistency is performed by the server before updating the database.

### **Access Control and Security**

The user is asked for her password before any data are accepted or delivered by the server. The authentication is valid through the entire TCP session.

If there are repeated attempts to get authorization with wrong passwords, *usermgrd* closes the connection after three of these attempts. If there are

more than 20 attacks within 10 minutes the server quits operation and sends mail with notice about its termination to root.

Passwords are never sent clear text over the net. The passwords are encrypted using a des binary (based on *libdes* [3]). If the des program could not be found, the client uses a very simple XOR modification to make it at least unreadable. We provide compiled binaries of the des program for:

- BSD/386
- SPARC
- Pmax
- HP/PA
- RS6000
- SGI

Whenever a new user is added the server will ask for a password for this user and do a lot of security checks on this password. It can even use programs like *checkpasswd* [4] to cross check the password against dictionaries. Only if it is a good password it will be accepted. The same procedure is used for changing passwords with *usermgr* or *rpaswd*. If you have good dictionaries, you might be well prepared for crack [5] attacks!

In case the perl library files *syslog.pl* and *syslog.ph* are available, *usermgrd* will log any interesting event with the syslog facility. In addition it keeps a log of events in a private log file.

### **Batch Processing**

Many sites have the need to add many users in a bulk process to their user name space. Instead of entering hundreds of users by hand, you can process the content of a file in batch mode using *usermgr*.

The entries for such a file can be generated by the perl script *account* which can be used as a login shell for a login granting account.

### **Customizing**

It is possible to extend the list of properties for a user. The server *usermgrd* uses a separate configuration file to learn about additional site specific user tags and their descriptions. This enables you to add as many features to a user as you like to.

### **The Targets**

The configuration process as performed by *GeNUAdmin* is divided into different targets. If a target should be processed, i.e., create configuration files and installation methods, it must be activated by setting a variable.

Here is a list of all currently available targets:

- |             |              |
|-------------|--------------|
| • Netgroup  | • Krb_Hosts  |
| • Passwd    | • Krb_Users  |
| • Ethers    | • Exports    |
| • Ypservers | • Fstabs     |
| • Group     | • Amd        |
| • Etchosts  | • Links      |
| • Dotrhosts | • Pseudonyms |

- Xaliases
- Boottab
- Nameserver
- Crontabs
- Distfile
- Resolv\_conf
- Inetd\_conf
- Syslog\_conf
- Services
- Dottwmrc
- Lpdhosts
- Printcaps
- Lhosts\_db
- Quota\_check
- Routing
- Ifconfig

Some of the targets are already described in the chapter about the databases ("The Databases"). Where it is useful an attempt will be made to group the descriptions for logically related targets.

## NIS

We can generate NIS maps [6] for the following files:

- passwd,
- hosts,
- group,
- netgroup,
- ethers,
- services, and
- ypservers.

As an alternative, local versions of these files can be updated if NIS is not used on a host.

## Local password files

There are two cases where it might be interesting to create local password files instead of using NIS.

1. You have machines that are not able to talk NIS.
2. You don't want NIS because of its security or reliability problems.

Each host can select – based on netgroups – which users are to be included in the local password file. There is the possibility to specify a minimal and maximal user id. Only users within that range will be taken from the central password pool. That is why local users like root, uucp, ... will not be touched.

## Name server

If selected, primary and secondary name server zone files and boot files are created including address, cname and mx records. Configurations for forward and reverse mappings can be generated. At some sites primary and secondary name servers with more than 20 forward and reverse maps are administered by GeNUAdmin.

## Sendmail

GeNUAdmin enables keeping up-to-date both IDA and V8 sendmail [7] configurations for a central mail server. The targets related to the mail system are:

- pseudonyms
- xaliases

The pseudonyms file is used by the central mail server to identify hosts for which it should accept mail.

The xaliases target installs an xaliases database for an IDA sendmail with incoming and outgoing aliases for all users. The aliases file contains user names and the users' full names.

## Printing

The contents of **printer.db** are evaluated to configure the printing system via the targets **printcaps** and **lpdhosts**. At the current point printer configuration is well supported for BSD-Systems. GeNUAdmin creates printcaps for servers and clients, creates spool directories, accounting files and logfiles.

The **/etc/hosts.lpd** files are created for all printer servers.

For some SYSV systems (hpux, riscos) setting up printing is also implemented. It might be easy to copy these methods to other SYSV systems.

## Filesystems

There are five targets concerning filesystem administration:

### *fstab and exports*

With the targets **fstab** and **exports** both files are created and installed and the current export and mount configuration is updated.

When creating the **fstab**, every effort is made to find the closest server – in terms of the network metric – of a set of replicated filesystems (e.g., **/usr/local**) or to find the closest interface name of multi homed servers for the simple reason that NFS and NIS always use the first address they get from the name services (NIS or BIND). This would cause all your NFS clients to use the same interface unless you explicitly tell them to use another interface.

### *amd*

It is possible to create **amd** maps for the Berkeley auto mounter **amd** [8]. The maps will contain filesystems of type **nfs**, **link** and **auto**. Replicated filesystems and multi homed hosts are also taken into account when creating **nfs** entries.

### *lngroups*

This is not really a target but included in the **amd** and **fstab** targets. Its purpose is to hide the real location of a users home directory from the users view and use something like **/homes/username** in the password file. The entry in **/homes** will be a soft link to the real location. This target maintains these soft links for all users on all machines.

It can be achieved by either using **amd**'s **link** filesystem in an **amd** map or by creating a script that makes the soft links for all users.

### *links*

This target evaluates the contents of the **links.db** database and maintains a set of soft links on all machines. The selection of soft links can be controlled using architecture, operating system type or host specifications.

### *quota\_check*

This creates quota lists for all local filesystems that use quotas. The lists are scanned by a utility perl script that will adjust the quotas of all users.

### Installing Users

If a user appears to be new in the user database, a script for installing a new user will be called. It is also possible to move the home directory of existing users from one disk to another by simply changing the home directory in the user database using *usermgr*. GeNUAdmin will recognize the new location and run the copy process.

### Kerberos

In the current version of GeNUAdmin only the administration of the host principals in the kerberos master database [9] is implemented.

For all hosts, the realm file called */etc/krb.realms* is generated and populated with all domains of the generic hosts. US export laws restrict us to Kerberos Version 4.

### Network Configuration

It is possible to generate a file that contains commands for creating static routes. This file can be executed at boot time if *routed* is not used to get the proper routing information.

It is also possible to create a file that contains commands to configure all available network interfaces. This, too, can be executed at boot time to configure all network interfaces.

### Daemon Configuration Files

The targets

- *Resolv*
- *Syslog\_conf*
- *Inetd\_conf*
- *Services*
- *Crontabs*

maintain consistent versions of a set of configuration files for system and network services. After installing updated versions, the corresponding daemons are notified about the new version.

With these targets a consistent central repository for the essential configuration files can be easily maintained.

### Miscellaneous

The following targets are intended to make your life as system administrator easier. Their main intent is to keep lists of hosts up-to-date with the current stock of machines.

*lhosts\_db*: This is a file which contains a list of all generic hosts with attributes like domain name, os-type and architecture. Using a little awk script, host name lists matching special properties can be retrieved from this database.

*distfile*: This creates a distfile for use with *rdist*. Host lists are provided for all hosts, all architectures, os-types and netgroups.

*dottwmrc*: This enables you to create *twmrc*, *fwmrc* and *mwmrc* files which contain menus with commands to open a window on a host. The menus can be grouped based on host netgroups.

*rhosts*: Entries for root's *.rhosts* file are created. For each host that should be included in the *.rhosts* file all interface names and all IP addresses are added since some systems – especially if they use NIS for host name lookups – will not recognize hosts if the connection does not come from the primary network interface.

*bootptab*: Creates bootptab files with entries for X terminals or PC's which use the bootp protocol for network booting.

### Technical Aspects

#### Running GeNUAdmin

There are several well separated steps in running a configuration process by GeNUAdmin.

- Dumping system data (password file contents, kerberos database, ...) into the GeNUAdmin workspace.
- Reading and consistency checking of the databases.
- Creating the configuration into a separate directory on the local disk without modifying the existing configuration.
- Installing selected targets on selected hosts or
- Installing the complete new configuration on all hosts.

It is possible to execute these different levels manually step by step and even redo them several times if needed. This enables checking of the intermediate results (e.g., the format and content of the new configuration files before installing them). There is also a command for saving an old configuration on the local disk and comparing a complete new configuration against a complete old configuration. This gives you an overview of the changes in all files of the configuration.

Usually, however, all steps will be executed with one command. This is normally done in regular intervals using the cron facility.

#### Implementation

GeNUAdmin's core part is written in perl and the utility programs are simple shell scripts. Hence porting to any flavor of a Unix operating system should be no issue as long as perl is available.

Operating system independent targets are included in the main driver script but all os-dependent parts are kept in separate files for each operating system type and target. The files are loaded at run time if they are available. A default method is provided for each target which can be used by operating system dependent routines. They might however implement a different method if the

default method is not applicable. If a method for an operating system does not exist, the target for machines of this type will be just ignored.

There is no problem in including non supported machine types into GeNUAdmin's framework. The os-type independent targets will be processed immediately and support for the machine dependent targets for the new machine may be added later.

Figure 6 shows an example of an operating system module for managing `inetd.conf` on an aix system.

```
#
# $Log: inetd_conf_aix.pl,v $
#
# $Revision: 1.2 $
# at $Date: 1994/05/02 $
#

sub inetd_conf_aix {
    local($h) = @_;
    &inetd_conf_default($h,
        "/etc/inetd.conf", "inetimp");
}
1;
```

**Figure 6:** An example module file for an operating system type (aix) depended method (`inetd_conf_aix`) which calls a default method (`inetd_conf_default`) with os-specific arguments. The specialty in this case is the third argument – the command `inetimp` – which exists and must be executed only on aix systems. Other systems will call `inetd_conf_default` with only two arguments and no command will be executed.

### Prerequisites

The prerequisites for using GeNUAdmin are minimal. You need two programs on each client for identifying the architecture (`arch`) and the operating system type (`ostype`) of the machine. The GeNUAdmin server needs `perl` (Version 4.036). On the client systems, `perl` is useful but not necessary. Some targets use lower quality shell scripts as replacement if `perl` is not available. Using `perl`, however, makes life much easier on the clients.

On each client which should have access to `usermgrd` using `usermgr` or `rpasswd`, `perl` is also needed unless the `perl` builtin `dump` function is used to create executable client programs.

GeNUAdmin uses `rsh` for the installation of the configuration and gets the data and programs from a read-only mounted NFS partition. An RPC based client-server model for exchanging the data and methods is under development.

### Related Work

GeNUAdmin borrows many ideas from systems like `Moir` from MIT [10]. By now, however, it has

grown a lot by adding many targets and tricks that we found necessary or useful at the sites, where we (GeNUA) are responsible for administration.

It still has some common functionality with systems like `Moir` from MIT or `Tivoli's Management Environment` [11]. There is however a major difference in the software and hardware requirements and the implications for your system when using one of the tools.

- No modification of the running system is needed. GeNUAdmin exclusively uses existing operating system tools and programs.
- No information about the internals of the running system or kernel is needed or used.
- You don't need any additional programs, servers, daemons, etc. on the clients save optionally `perl`.
- It's written in a script language (`perl`) and can easily be modified, extended and ported to new unix platforms.

### Future Plans

- The distribution and installation of the configuration should be done using a RPC based client/server method instead of `rsh/NFS`. All data transmission will then be done via a TCP socket. The need for transmission could be determined using checksums to minimize network traffic.
- Support of AFS and DCE environments will come, too.
- An interface to high performance databases for the storage of large amounts of data instead of using flat ASCII files should be implemented. We plan to support first the freely available `postgres` database [12].

### Availability

GeNUAdmin is available without charges. Contact the author for further information.

Most of the targets from section "The Targets" are implemented on the following operating systems:

- SunOS,
- Ultrix,
- HP-UX,
- IRIX4/5,
- AIX-3.2.x,
- DEC-OSF1,
- BSD/386,
- Solaris, and
- RISCOS.

### Author Information

Magnus Harlander received his diploma and doctoral degree in theoretical physics from the Technische Universität München (TUM), where he had been working in theoretical nuclear physics and in the field of neural networks. He has been

responsible for system administration at the TUM and at Lawrence Berkeley Lab for several years until he became a co-founder of GeNUA (Gesellschaft für Netzwerk- und Unix-Administration) in 1992. Within his company he is responsible for software development. Most of these projects are targeted towards GeNUA's main field of activity: Unix System Administration. He can be reached via e-mail at [harlan@genua.de](mailto:harlan@genua.de).

### References

- [1] Larry Wall, Randal L. Schwartz, *Programming perl*, O'Reilly and Associates, Sebastopol, CA, 1991.
- [2] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley Publishing Company, Inc, ISBN 0-201-63337-X), 1994.
- [3] Eric Young, *libdes* Version 3.00, available via anonymous ftp from e.g., from [ftp.psy.uq.oz.au](ftp:psu.uq.oz.au), 1993.
- [4] Clyde Hoover, *npasswd* and *checkpasswd* from the University of Texas, available via anonymous ftp from [ftp.cc.utexas.edu](ftp:cc.utexas.edu), 1990.
- [5] Alec D. Muffet, *Crack the Sensible Unix Password Cracker*, available via anonymous ftp, e.g., from [ftp.informatik.tu-muenchen.de](ftp:informatik.tu-muenchen.de).
- [6] Sun Microsystems Inc. *Network Information Services* SunOS Reference Manual, Volume 1 Section 8: *ypbind(8)*, *ypserv(8)*.
- [7] B. Costales, *sendmail*, O'Reilly and Associates, Sebastopol, CA, 1993.
- [8] Jan-Simon Pendry and Nick Williams, *The 4.4 BSD Automounter*, Documentation to amd version 5.3, available via anonymous ftp from [usc.edu](ftp:usc.edu) in the directory */pub/amd*, 1991.
- [9] J.G. Steiner, C. Neuman, J.I. Schiller, *Kerberos: An Authentication Service for Open System Networks*, Usenix Conference Proceedings, Winter 1988, pp 191-202.
- [10] Mark A. Rosenstein, Daniel E. Geer, Peter J. Levine, *The Athena Service Management System*, Winter Usenix 1988.
- [11] Tivoli Systems, *Tivoli Management Environment*, 1992.
- [12] M. Stonebraker, *Documentation to Postgres 4.2*, available via anonymous ftp from [postgres.cs.berkeley.edu](ftp:postgres.cs.berkeley.edu).

# Config: A Mechanism for Installing and Tracking System Configurations

John P. Rouillard and Richard B. Martin – University of Massachusetts at Boston

## ABSTRACT

One problem that faces system administrators is how to install and maintain local configuration information on a large number of machines. Some previous approaches such as cloning [2] help, but they only provide a baseline, not ongoing configuration control. Other mechanisms such as Typecast [7], Hobgoblin [5], Scrape [3] or Mkserv [6] assist in the configuration process, and provide some support for ongoing maintenance. However supporting multiple system configurations is still troublesome. Also it can be very difficult to delegate system administration tasks. Insufficient logging of file changes can create a nightmare when attempting to find the cause of a problem.

The method we present uses *rdist* and integrates it with *make*(1) and the CVS version control system to provide the ability to delegate and log changes. End node users can make changes to their workstations, however all changes are logged, so that it is possible to see what has changed on a given machine when problems occur. When making changes that affect a large number of machines (e.g., amd automount maps, rc files) previous versions of the file are available in the CVS tree and can be retrieved and distributed in case of unforeseen problems.

## Introduction

The University of Massachusetts at Boston has used *rdist* from a single configuration area to ease multiple machine maintenance. While under contract to Siemens Nixdorf Research and Development in Burlington, MA<sup>1</sup>, John had a chance to further refine the *rdist* system to add version control elements to the system. As an added feature of this merger, delegation of responsibility for changes to a file is feasible.

The main features provided by the config method are:

- Ability to track changes to any and all configuration files and binaries on a system.
- Files that are to be distributed can be required to pass sanity checks to reduce the chance of errors.
- Concerned parties can be notified of file changes via electronic mail, news, or real time messaging such as alphanumeric pager, inform, or zephyr.
- Responsibility for changing files can be delegated to end users with fine grained access control.
- Multiple authorized people can change configuration files without conflict.
- Retrieval of previous (read working) versions of files allows changes to be backed out easily.

The main programs that are used are the CVS source code control system, *rdist* version 6.1, GNU *make* and a home grown perl script that generates *rdist* macro (class) definitions from a database file.

The config system embodies all local configuration information for a host. Any system file on the workstation that is different from the file in a stock installation is included in the config system. This includes modifications to rc files, the fstab and vfstab files, the password file, as well as binaries that are required for operation (e.g., the amd automounter).

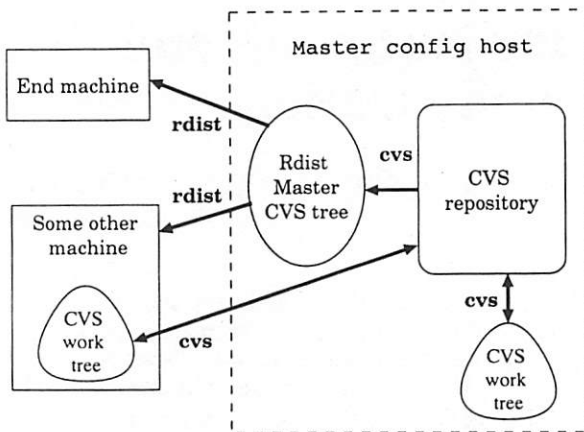
The implementation described in this paper is based on *rdist*, and therefore uses a push paradigm, but other distribution mechanisms such as *track*, *sup*, or *rcp* could be used.

## The Components

Figure 1 displays the primary components of the system and the relations between them. The first component of the system is the "Rdist Master CVS Tree" hereafter referred to as the *rdist* master tree. This tree contains all local configuration information for a host.

In effect, the *rdist* master tree contains all of the information needed to tailor the vendor's distributed filesystem hierarchy to local requirements. This information embodies all of the "personality" of the workstation. Distribution to the workstation is controlled by a master distfile that resides in a sub-directory of the *rdist* master tree.

<sup>1</sup>Primary extension development was done there, some other features were added at other sites.



**Figure 1:** The components of the config system. The heavy lines show the update mechanisms, the programs used for the update are printed in bold face type near the update lines. The arrow heads indicate the directions of update that are permissible.

The rdist master tree represents the head revision of every file in the CVS repository<sup>2</sup>. The rdist master tree is updated from the CVS repository before an rdist is done. While hand editing of files in the rdist tree is possible, distributing these modified files requires circumventing the standard distribution mechanism, and thus is done in only exceptional circumstances.

There are two program components for a user of the config system:

- A Bourne Shell wrapper program for rdist version 6 that performs the underlying updates and consistency checks for the rdist master tree. This script is called capital “R” – Rdist.
- CVS version 1.3 is used to check modified files into the CVS repository.

In addition to the two programs above, some mechanism for allowing the Rdist script to be executed as root must be provided. We use sudo [4] version 1.3.

The programs that are used by the system itself are not that much more complex, but there are more of them.

- rdist [1] version 6.0 or newer. While config can be implemented with earlier versions of rdist, there is additional administrative overhead and performance/logging issues to consider.
- CVS is used to update the rdist master tree from the CVS repository.
- Gnu make is used to generate machine and

group specific files from the sources maintained under CVS control.

- A perl program collects data from a simple database file and emits this data as a set of rdist macros (classes) for use in a distfile. This tool is called `class_gen`, since its original purpose was to generate classes for rdist. It has since been modified to produce `/etc/hosts` style output as well as to produce a list of host names matching various criteria.

### The Distribution Mechanism, rdist and Friends

This section covers the end node distribution mechanism. This encompasses, rdist version 6, the Rdist shell wrapper, the Distfile and the arrangement of the rdist master tree.

#### rdist Version 6

The config system is based on Rdist version 6.0 or newer. This version of rdist has a number of features that enhances the distribution portion of the config system:

- Multiple machines can be updated in parallel. This dramatically speeds update performance.
- Set operations such as union, difference and intersection can be performed between sets of hosts. This makes it much easier to rdist files to all hosts except a few hosts.
- More information is logged. Logging using syslog is supported.
- Handling of downed or non-responsive hosts is much improved.
- It is portable to many platforms.
- With version 6.1 rdist no longer needs to be setuid root. It can also use more secure communications channels such as a Kerberized rsh.
- A single command can be run after all files in a rdist command have been distributed.
- Changed versions of files can be saved before being overwritten.

Because of rdist's parallel update capability this system scales well to a few hundred hosts. When scaling to a very large site, multiple rdist master trees on multiple hosts and a hierarchical update cascade mechanism should be used to facilitate timely updates.

#### The Rdist Wrapper Script

The rdist program is called from a Bourne shell script wrapper called Rdist, which performs a number of functions:

- Generates a list of directories that are to be distributed<sup>3</sup>, and invokes “cvs update” on each directory to pick up the most recent

<sup>2</sup>Some end files are automatically generated from template files, and only the template files and the mechanism for generating the end files are kept under version control.

<sup>3</sup>As you will see in section “The Distfile and the rdist master tree” there is usually a one to one correspondence between distfile targets and the directories in the rdist master tree

changes to the config tree. It also verifies that no changes have occurred to files in the rdist master tree. If changes are found in the rdist master tree that are not in the CVS tree, the script aborts.

- Looks for makefiles in the directories that are to be distributed, if a makefile is found, it runs *make*(1) in that subdirectory.
- Determines what hosts are down (using *rup*time or *luptime*) and removes them from the host list that it passes to *rdist*. This is a hold-over from *rdist* versions before 6.0 that used to hang forever on downed hosts, but it is still a useful function.
- Creates the master Distfile on the fly from a template, and runs *rdist*. Optionally it passes the output of *rdist* through summary filters that make viewing the output much easier.

#### *The Distfile and the rdist Master Tree*

The *Rdist* shell script couples the distfile to the structure of the *rdist* master tree, so it makes sense to cover both of these items in the same section.

The distfile that is used by *config* has four target types. The first target type we call a "standard" type. A standard type target is labeled in the Distfile with a string with a lower case initial letter. Every top level subdirectory of the *rdist* master tree that starts with a lower case letter has a corresponding target in the distfile. Using the directory names as targets provides a nice update paradigm of:

```
% cvs co crontabs
% cvs ci crontabs
% sudo Rdist crontabs
```

---

```
binaries:
/config/Bindist/yppasswd.save -> (${IRIX_HOSTS} ${RISCOS_HOSTS})
    install -onumchkggroup /usr/bin/.yppasswd.save ;
    special /config/Bindist/yppasswd.save "/bin/sh \
        /usr/bin/.yppasswd.save";

dots:
/config/dots/{.forward,.rhosts} -> ${ALL_HOSTS} - ( ${BSDI_HOSTS} ra )
    install / ;

hosts:
/config/hosts/hosts -> (${ALL_HOSTS})
    install /etc/hosts;

kernel:
/config/kernel/sinix/stune -> (${SINIX.D_HOSTS}) & (${afs_C_HOSTS})
    install /etc/conf/cf.d/stune ;
    special /config/kernel/sinix/stune "/etc/conf/bin/idbuild";

sendmail:
/config/sendmail/ida.cf/mailhost.cf -> (${MAIL_HOSTS})
    install -b /etc/sendmail.cf;
    special /config/sendmail/ida.cf/mailhost.cf
        "/usr/local/etc/restartmail";

sendmail:
/config/sendmail/ida.cf/mailslave.cf -> (${MAIL_SLAVES})
    install -b /etc/sendmail.cf;
    special /config/sendmail/ida.cf/mailslave.cf
        "/usr/lib/sendmail -bz";

tcpd_inst:
/config/tcpd/ultrix.bin/{tcpd.install,tcpd} -> (${ULTRIX_HOSTS})
    install -b /usr/etc ;
    special /config/tcpd/ultrix.bin/tcpd "/usr/etc/tcpd.install dec" ;

xntp:
/config/xntp/dlws9 -> dlws9.me.com
    install /etc/ntp.conf ;

xntp:
/config/xntp/dl5000 -> dl5000.me.com
    install /etc/ntp.conf ;
```

Figure 2: Sample entries in the distfile.base base distfile

When Rdist is invoked without specifying targets on the command line, it generates a list of targets for passing to rdist using `echo [a-z]*`. Thus an Rdist without any explicit targets distributes all of the standard targets. This operation is done nightly out of cron. We have come to count on the nightly Rdist cleaning up experimental versions of configuration files (e.g. `inetd.conf`, `services`) that may be changed for a short term test, but that are not supposed to be permanent changes.

Some "standard" targets are actually dummy targets, consisting of a Makefile that forces a CVS update of appropriate files before rdist is invoked. While use of such dummies is discouraged, there are times when they are useful when delegating responsibility for file modification.

The second type of target name is a subelement type. These appear as `<subdir>.modifier`. These cause the `<subdir>` to be updated as described above. It is policy that these subelement types distribute a subset of the files that the regular `<subdir>` target would distribute.

The third type is an install type. These are of the form `<subdir>_inst`. These cause the corresponding standard target directory (i.e., `<subdir>`) to be updated before a distribution occurs. These are reserved for single shot operations that are done when initially installing a machine. For example initial kernel config files that are expected to be modified by a later package installation, empty directory creation or wrapper program installation with special commands that are best not repeated.

The fourth type of target is called an "action" target. These targets are similar to the install targets in that some action is performed that may not always be necessary. One example of an action target is the `rc.reboot` target. As you might guess, this target distributes the `rc` files, and then forces the machine to reboot.

The Distfile is generated from the output of the `class_gen` script and the file `distfile.base` using the Makefile in the `Db` directory. The `distfile.base` is what most people think of as a distfile. Figure 2 shows some typical entries.

Actual host names are almost never put into `distfile.base`. Instead canonical macros are used in place of host names. These canonical macros are produced by the `config_class` program using the information in the database file. `Config_class` and its database file are described in the database section.

#### The Rdist Master Tree

The structure of a typical rdist tree is shown in Figure 3. The `Db` directory contains various administration files. As the name `Db` implies, there is a simple database file contained in the `Db` directory. Originally that was all that was in the `Db` directory, however as the system matured, the `Db` directory

was used for general administration files, and file generation. Thus a better name for the `Db` directory would be `Admin`.

A perl script creates rdist macro definitions that are prepended to the base distfile to create the Distfile that is used by rdist. The Makefile in the `Db` directory is invoked by the Rdist wrapper script and makes certain that the Distfile is brought up to date.

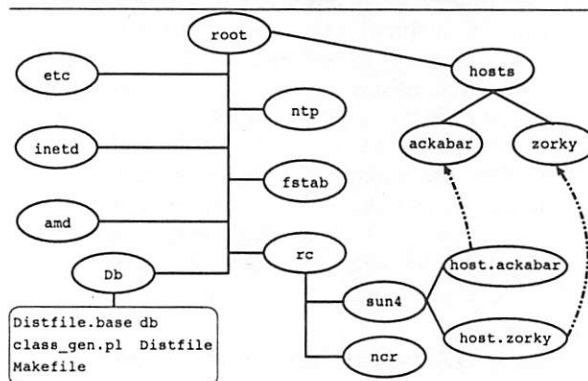


Figure 3: A typical portion of a CVS file tree repository showing some of the contents of the `Db` database directory.

In Figure 3, directories (i.e., standard targets) are represented by ovals. The directory "hosts" is special. It represents files that have been delegated to the owners of those hosts. The "hosts" subtree has subdirectories named after the hosts which receive delegated files. Each subdirectory has a structure that is identical to the structure of the rdist master directory, but is writable by the delegates. The separate subtree for the delegates files is required by CVS's check-in mechanism. Write permission on the repository directory is required for CVS check-in, but it is not appropriate that delegates have write permission on rdist master directories.

The segregation of files in the "hosts" tree produces a slight problem. When we type "Rdist rc", we want all `rc` files pushed to all hosts. However, when CVS runs down the `rc` directory tree to update all of the files, it will miss the files in `hosts/*/rc`. We could rewrite the Rdist script to run down the "hosts" tree and start CVS update in the appropriate directories, but John chose another method. Each subdirectory under the host specific tree is linked into the main tree. These links are indicated by the dashed lines in the diagram. CVS traverses the links and automatically updates the files found there. The links are automatically created by a script run on checkout for files in the "hosts" tree.

#### CVS

CVS is somewhat different from other well known version control programs in that it supports parallel development with a merge operation at check-in rather than the serial locking paradigm supported by SCCS and RCS. Due to this parallel

development ability, multiple system administrators can make changes to a given file, during a crisis, without wasting time trying to find out who has the file locked. This is very important in an interrupt driven job such as system administration, since it is impossible to tell in advance just what files may need to be modified in a given day. Most configuration files are text files, or are derivable from text files, so CVS's line by line conflict resolution mechanism works very well and allows small simple changes to be done to the configuration files while other longer running changes are still in process. CVS's conflict resolution mechanism does not work well on binary files. In our experience binaries files change infrequently. Changing binaries is not a "quick fix" type of operation, so the delay inherent in a serial locking paradigm is not a problem. We use the hooks in CVS to provide a serial locking paradigm when changing binary files.

CVS actually uses RCS internally, and thus provides most of the functionality of RCS such as named version information (useful for making checkpoints on the first of the month). The more mundane, but crucial aspects of a version control system are also provided. Namely, all previous versions of the files are available from the RCS configuration files and the names of the people who made changes are recorded for posterity.

One additional and crucial advantage of using CVS as opposed to RCS is its handling of log information. The log information can be routed to people through various means. News and email have been the standard mechanism up to now, but one site where config is deployed has been using alphanumeric pagers and the inform system for real time notification of changes to the CVS tree. One additional feature of the the CVS log facility is that different people can be notified about changes in different parts of the tree. So a change made to files for the host ackbar, can trigger email to the person who is supposed to be maintaining that system.

Recently CVS's ability to run verification programs on check-in has been exploited. If any of these verification programs exit with a non-zero status, the check in is aborted. These programs are being used to provide sanity checks on data files, check to see if particular files are locked, or make sure that the person attempting to make the change has permission. For instance the amdxfref program was used to validate amd automounter maps during check-in to reduce the chances of an incorrect map being checked in. Sanity checks for hosts files, and inetd.conf files have also been implemented. Also `sh -n` makes a pretty good sanity check for shell scripts such as rc files.

#### Gnu Make

We use GNU make in preference to other makes because it provides some very useful features

(e.g., automatic makefile generation, and tracing ability). Also it is available across a large number of platforms making the rdist master tree platform independent. This allows rdist master trees to be scattered across multiple platforms which is important for scalability, and may be required for availability. While the use of make is not required, it provides each standard target with its own method of generating host or class specific files. These methods can be as trivial or complex as needed. These methods ease the task of maintenance especially when there are only minor differences among the various files.

As part of the Rdist wrapper, each top level directory is checked for a makefile. If one is found, a make is done in that directory. These inferior makes can be used for a number of purposes. One purpose is to generate host specific files such as `inetd.conf`, or `crontab`. Using `cpp` and `sed`, different version of the files can be generated from a generic template file. These generated files are distributed via rdist. Using prototype files in this way greatly reduces the amount of work that must be done when performing a global change such as installing a new housekeeping program, or adding a new experimental service.

#### The Database System

A simple database is used to record essential information about the hosts handled by the config system. This information includes: the machine name, the machine's operating system and version information, its administrative group and the services it provides and uses. A perl script, called `class_gen`, creates rdist macro definitions that are prepended to the base distfile. This composite Distfile is used by rdist. The Makefile in the Db directory is invoked by the Rdist wrapper script to make certain that the Distfile is kept up to date.

#### The Database File

Figure 4 shows two sample database entries. The format of the entries in the database file is a simple `keyword = value` syntax. A host entry starts with the "Machine" keyword and continues until the next "Machine" keyword. The keywords: `cluster`, `os`, `patches`, `serves` and `uses` are used by the `class_gen` program to create classes of host for rdist. To ensure that the data in the database is up to date, simple scripts have been written that verify information such as the `os` and `os revision` and the amount of disk and memory.

When first installing a machine, the appropriate information is set up by hand. A special version of the Rdist script, called `Install`, is used to bring the newly installed machine up to date with the master rdist tree.

```
Machine = dl5000.me.com
Alias = dl5000 nexus master
arch = D5000/200
cluster = bedford
ip=132.121.14.10
enet= 00:00:08:04:05
os = Ultrix 4.2
serves = jukebox decnet bind time
serves = mail pci lat print_lp
serves = syslogm fingerm
uses = network
patches = 002-34A 005-153C 82352-bc
pmemory = 48M
kernel_name = nexus
```

and

```
Machine = amethyst
alias = am
ip = 136.121.32.20
cluster = dev
arch = R4000
cpu = R4000
pmemory = 32M
disk = 300M
owner = gink
# os = type and version
# with .'s between
# revision steps.
os = IRIX 4.0.5.f
```

Figure 4: Two sample database entries.

```
ALL_HOSTS=( amethyst chicago claudé
god iris orchid ra violet )

IRIX_4.0.5=( iris violet )
IRIX_4.0.5.F=( amethyst )
IRIX_4.0.5.X=( amethyst )
IRIX_4.0.X=( amethyst iris violet )
IRIX_4.X=( amethyst iris violet )
IRIX_5.0=( orchid )
IRIX_5.X=( orchid )
IRIX_HOSTS=( amethyst iris orchid
violet )
DOS_3.1=( god )
DOS_3.X=( god )
DOS_HOSTS=( chicago god )
es_C_HOSTS=( ra claudé iris )
SYSLOGM_HOSTS=( ra )
BIND_HOSTS=( ra claudé )
```

Figure 5: Representative output from the class\_gen program when used to generate rdist macros.

#### Class\_gen

Sample output from class\_gen is shown in Figure 5 and includes four types of output:

- Global output. This is shown by the ALL\_HOSTS macro.
- Host os/version. Multiple macros are created

for each os/version item in the database file. There is one macro created for every os version, and every os parent version. This cascade can be seen with IRIX hosts. Not to be left out, DOS as well as all other host types can be represented.

- Cluster groups. It is often useful to be able to group machine by task, or by the workgroup to which they are assigned. In figure "Class\_gen" the Engineering services cluster hosts are listed with the canonical name es\_C\_HOSTS.
- Hosts that serve particular information for example they are syslog master servers, or supply bind (DNS) service have their own groups.

In addition to the macros shown in Figure 5, the database keywords: "uses" and "patches" produce their own classes in much the same way that the "serves" keyword produces classes.

#### Tasks

The config system gets a host into a known state automatically and repeatably. This ability eases a number of tasks.

#### Installing/Upgrading a System/Crash Recovery

The process of installing a new system, of a type already handled in the config system, starts with a stock os installation. After the installation has been done, and the machine is on the network, add an entry to the database file, and run rdist against the machine until no more files are transferred. Then reboot the machine and it should be fully configured.

Crash recovery is also greatly simplified since there is no need to go to backup tapes to reload the base machine configuration after installing the base system.

Updating the operating system is handled the same as installation because we don't trust the "Fast Upgrade" procedures that vendors come out with. Also, an upgrade is a good excuse to clear out all of the cruft from the filesystem.

#### The Update Procedure

Updating a file under CVS control is very easy.

- Check out your own local copy of the file in question using "cvs co".
- Edit the file appropriately.
- Perform a test install of the file to make sure that it works as expected. [optional but suggested].
- Check in the new file using "cvs ci".
- Distribute the file using Rdist. If you didn't perform a test install, it is advisable to rdist to a subset of hosts for testing purposes before distributing to the whole network.

We find that people are usually very good about testing their changes before inflicting them on the world since their name is associated with the

check in notice that is distributed via email, and news. If they aren't good about it, one or two mistakes later, they are very good about it. It is recommended that sanity checking programs be crafted for the files since using this feature makes it much safer to have novices changing files. At the very least the sanity check programs will prevent catastrophic errors from being checked in. When you have novices changing files, sanity checking programs<sup>4</sup> can be considered mandatory.

### Creating New Targets

When creating new targets, the most difficult question is where to put them in the config tree hierarchy. Targets that are likely to be changed often should be near the top of the tree to make it easy to remember the target names, and to make the target name (or CVS module/directory name) easy and fast to type. It also makes it easier for new system administrators to develop a feeling for the most heavily used files, and thus the most active tasks.

In addition to structuring your tree by function (e.g. directories for passwords, patches, etc) it is often useful to create directories that mimic the tree structure where the files would be found. For example the /etc/services file has the same structure on all hosts. It is usually an easy matter to produce a single services file that will satisfy all of the hosts in the network. Files like these can often be placed into a directory such as "etc".

Some files such as fstab/vfstab files aren't often changed, but it seems that each architecture has its own variation on the contents of these files. A reasonable location to put these files would be in etc/fstab, and populate the directory with the vfstabs or fstabs for the different architectures, or machines. A subelement target (e.g., etc.fstab) could be used to force distribution of just the fstab/vfstab files.

We have found that the subelement targets are rarely used in practice. In most cases, the encompassing simple target is used instead. The only exceptions to this are for those simple targets that use lots of data. For example, the target "Patches.solaris" exists in both the CVS module definition files and in the distfile. This reduces the data that must be checked out of the Patch directory when updating the solaris patch tree. There is no need to maintain a one to one correspondence between CVS modules and the subelement targets in the distfile, although such a coupling can be made. The implicit relation that the non-standard targets have to their standard targets provides the coupling into the CVS config tree; it is only at this level that the CVS tree and the distfiles are tightly coupled.

One additional feature of config, is that you don't have to start using all of the features of config

at once. When creating a target, you can start out with a simple target/directory that has one file for each host enrolled in config. This would produce a large distfile, but it would be simple to understand, and very little could go wrong.

As you tire of making the same changes to multiple copies of the same file, the various classes that are created using the database file, and the class\_gen script can reduce the needed changes to files that actually have different contents. If even this reduction in work is not sufficient, the ability of config to run a makefile to generate the individual class files from a template can be exploited. Auto-generation of files can fail, incorrect database entries, disk full conditions, etc. can interfere with the proper operation of the system, but we have found that these failure modes occur infrequently and are easily diagnosed and fixed.

### Delegating Responsibility For Files

When you have to delegate responsibility for files there are a number of methods by which access can be granted. The easiest is to use RCS's own access list policy to restrict the people who can access a file. Alternatively or in addition, other access checks can be performed by the programs that are run upon a CVS check-in. Once access is granted via whatever means, and suitable sanity checking programs are put in place, it is very easy to allow people to change files.

However, it is critically important that the master distfile.base is kept under tight control. This way the person is unable to "delegate" himself responsibility for extra files, since these extra files won't be propagated to the end node machines.

Also, it is important to prevent the novice system administrator from running anything but "standard" rdist targets since install and action targets can cause actions to occur that would be detrimental to host operation.

### Fixing Problems with Hosts

When a problem is encountered with a host, the first step is to run Rdist against that host to make sure that it has all of the proper files in place. In the past many problems have been handled simply by distributing the known configuration files onto the remote host, possibly followed by a reboot.

Sadly not all problems can be solved so easily. This is where the CVS component of the system shines. The question that is invariably asked is: "What has changed since *date* when everything was working?". There are a couple of ways of answering this question. If you have been using CVS's logging ability to its fullest, there is a newsgroup, or a mail file folder that has a list of all logs that have been produced. Then is it a simple matter of looking at the logs after a particular date.

<sup>4</sup>For the files, if not the people.

Another way to answer the question is to use the rdist master tree itself. A cd to the top of the rdist master tree followed by a "cvs log -d date" command shows all of the log information on files checked in since the specified date. Sadly, the RCS rlog command also lists header information for files that do not have new revisions in the date range specified, but this is easily handled using a wrapper script that filters out the unneeded information.

One additional function that was not originally envisioned was to link the CVS log reports into a trouble reporting system by using a site specific script to filter the log. Using this method, changes recorded under CVS can be automatically entered into a trouble reporting system, and can be linked to the corresponding trouble ticket. This provides an easy mechanism for formally documenting the changes necessary for a problem's solution.

With change information in hand, the job of analyzing the cause of the problem is often greatly simplified.

### File Scanning

It is possible that a change is made to files that are not under the config system. In this case, the host can not be recreated from the os installation tapes, and the information in the rdist area. Troubleshooting problems under these circumstances is difficult if the cause of the malfunction is this uncatalogued file.

A system integrity checker is used to discover these unenrolled files. Since we use tripwire to check system integrity for security purposes, it was only natural that it also would be used for checking the integrity of the entire system. For the purpose of insuring integrity for config, a simple checksum such as crc-16 or crc-32 can be used saving a lot of time that is needed to compute more sophisticated checksums such as MD4 or Snefru. We have had good luck with using tripwire in this "simple" mode to discover files that should have been put under config control, but weren't.

### Conclusion

This system has been used to provide support from networks of 10 machines with two administrators to 280 machines with 10 administrators. We have found the above system to be very flexible and easily extendable to support many different notification mechanisms. Using multiple rdist master trees on different hosts, and keeping a duplicate copy of the CVS master tree on a second rdist master host allows the system to be quite robust in the face of network interruptions and system downtime.

However all is not perfect. There are a few areas that we will be concentrating on for future development. As the system currently stands, it is possible for config users to circumvent the logging mechanism by directly modifying the CVS version

control files. This problem can be minimized or potentially eliminated by running CVS and the underlying RCS programs setuid. In practice we have found this to not be a problem since updating and distributing files using the standard mechanism is much easier than trying to change the CVS repository files by hand, however some provision for supporting hostile users must be made.

A second problem stems from the system's flexibility. Because the system is so flexible, it is easy to create a config tree structure that makes it difficult to find the files that you want to change. RCS \$source:\$ strings mitigate the problem, but they are not definitive: automounting and symbolic links can confuse the issue. Also as the config tree evolves, CVS's lack of support for renaming complicates rearrangement due to paradigm shifts.

### Availability

The class\_gen perl script and the slides for this talk will be available from anonymous ftp site ftp.cs.umb.edu in the files:

```
/pub/bblisa/talks/config/config.tar.Z
/pub/bblisa/talks/config/config.slides.tar.Z
```

Rdist version 6.x is available from usc.edu, CVS version 1.3 is available from UUNET and was submitted to comp.sources. The master Rdist script is available but note that it is so crufty, that it is getting unmaintainable, and needs a complete rewrite.

### Author Information

John Rouillard graduated from the University of Massachusetts at Boston with a B.S. in Physics in 1990. Since then he has been a contractor specializing in tool building and automation of various system administration tasks. In January of 1994 he took over release engineering and development from Brent Chapman for the Majordomo mailing list management tool. At the same time, he became a Senior Systems Consultant for the Mathematics and Computer Sciences Department's Software Engineering and Research Labs at the Univ. of Massachusetts at Boston where he continues his system administration automation tasks for various Lab clients.

Richard Martin attended Merrimack College and UMass-Boston. He has been System Programmer with the department of Math and Computer Science since 1984, maintaining hardware and software and training system administrators.

### Bibliography

- [1] Cooper, Michael. "Overhauling Rdist for the '90s", LISA VI proceedings, October 1992, pp 175-188.
- [2] Jones, George M. and Steven M. Romig, "Cloning Customized Hosts (or Customizing Cloned Hosts)", LISA V proceedings, September 1991, pp 233-241.

- [3] Kint, Richard W. "SCRAPE (System Configuration Resource and Process Exception) Monitor", LISA V proceedings, September 1991, pp 217-226.
- [4] Nemeth, Evi and Garth Snyder and Scott Seebass, "UNIX System Administration Handbook", Prentice Hall, 1989.
- [5] Rich, Kenneth and Scott Leadley. "*hobgoblin*: A File and Directory Auditor", LISA V proceedings, September 1991, pp 199-207.
- [6] Rosenstein, Mark and Ezra Peisach, "Mkserv - Workstation Customization and Privatization", LISA VI proceedings, October 1992, pp 89-95.
- [7] Zwicky, Elizabeth D., "Typecast: Beyond Cloned Hosts", LISA VI proceedings, October 1992, pp 73-78



# Towards a High-Level Machine Configuration System

Paul Anderson – University of Edinburgh

## ABSTRACT

This paper presents a machine configuration system which stores all configuration parameters in a central “database”. The system is *dynamic* in the sense that machines reconfigure themselves to reflect any changes in the database whenever they are rebooted. The use of a central database allows configurations to be validated, and correct configurations to be automatically generated from policy rules and high-level descriptions of the network. A permanent record of every machine configuration is always available and the system is extensible to handle configuration of new subsystems in a modular way. The paper includes a review of previously published work and common techniques for *cloning* and configuring workstations.

## Introduction

When a new machine is installed, it will rarely be used with the default configuration supplied by the vendor of the operating system. The partitioning and allocation of space on the disks, the software packages to be carried, and the network name and address are typical *configuration parameters* that will be set differently by different sites and for different machines at the same site. In addition to these basic parameters, most large sites will require a more extensive customisation of the basic system, for example running additional or replacement daemon processes such as time synchronisation.

Most vendors provide some kind of installation procedure which allows the basic configuration parameters to be set. However, in a typical large site, these procedures are nearly always inadequate for one or more of the following reasons:

- The procedures cover only the vendor-supplied software and are not extensible to cover local and third-party software.
- The interface to the procedures is often a GUI and cannot easily be automated for handling large numbers of systems.
- The procedures are not complete, and further manual operations (for example, `crontab`), or additional hand-editing of configuration files (for example, `inetd.conf`), are required to completely configure the machine.
- The configuration information is stored on the machine itself so that it must be re-entered whenever the machine is re-installed, and it is unavailable for inspection when the machine is down.
- The procedures are highly vendor-specific and are not appropriate for use in a heterogeneous environment.

Sites with a small number of machines, or simple configuration requirements, sometimes use only

the vendor-supplied procedures, but this means that machine upgrades or installations require considerable manual intervention. Large sites will usually have developed their own procedures to help overcome some of these problems, and the following section surveys some of the techniques that have been used.

The remainder of the paper describes a configuration procedure that has been developed for use in the Computer Science Department at Edinburgh University. This stores complete machine configuration information in a central “database”, allowing configurations to be validated and automatically generated. The system is also modular so that new subsystems can be added independently to the configuration procedure.

## Background

Most vendor-supplied installation and configuration tools suffer from all of the problems listed in the previous section. In many cases, attempts to simplify installation for small sites (for example, graphical user interfaces) have caused further difficulties for large sites. Even where some provision has been made for large-scale automation (such as Sun *auto-install*[1]), the configuration process is still inadequate for the other reasons given above.

The most common technique for dealing with a large number of machines is *cloning*. Cloning procedures are not normally supplied by the vendor, but different systems have evolved at many large sites (for example, Ohio State University[2]), all sharing similar characteristics. A single *template* file-system is hand-crafted with the site-specific configuration information and replicated directly to create a new machine. Clearly, such a pure cloning process is only sufficient if there are no machine-specific configuration parameters, and every machine on the site has an identical basic file-system (or there are

a small number of categories). This approach has been taken in some cases, such as the Athena[3] system, but it usually requires unacceptable modifications to the vendor's base operating system.

Various schemes have been used for applying machine-specific changes to the template after (or during) the cloning operation; for example, the above Ohio scheme, `typecast`[4] and `mkserv`[5]. These are adequate for environments where the configurations are largely static and similar. However, they can become unwieldy when there is a wide variation in the required configurations and/or frequent changes. It can often be difficult to determine the configuration that is actually being applied to an individual machine; in some cases, this information might not exist explicitly<sup>1</sup>; in other cases, it might exist in a wide range of different files and formats. The lack of modularity in the configuration process also makes it difficult for different people to maintain the configuration of separate subsystems, and changing the configuration of an existing machine is usually difficult.

Storing the machine-specific configuration information explicitly in some external database (for example, `sad`[6]) is a major improvement, since the configuration of a particular machine is always clear and the information is always accessible, even when the machine is down. There is still the option of using procedural rules to generate certain configuration parameters<sup>2</sup> but the rules are evaluated before the machine is actually configured and the results of the evaluation are visible explicitly in the database.

The information from such a database can be used during the cloning process to control the creation of the file-systems when the machine is being built. In this case, the machine-specific characteristics are hard-wired into the file-system and the database information is no longer required for the running of the machine (a *static* configuration). Alternatively, all machines can be created as pure clones and the configuration information can be read *dynamically* from the database as the machines are running (usually at boot time). If the configuration information is used in a static way, it is difficult to change without completely re-cloning the system, but the machine is not dependent on the availability of the central database, and no configuration procedures need to be run at boot time. Dynamic configuration requires special configuration procedures (usually run at boot time) and the machine is dependent on the existence of the central database,

<sup>1</sup>For example, a particular configuration parameter might be generated "on the fly" at installation time by a script which implements some kind of policy rule.

<sup>2</sup>For example, there may be a rule of the form "machines belonging the research group always carry GNU Emacs".

but it does allow changes in the database to be reflected immediately in the actual machine configuration.

A purely dynamic system is normally impractical for several reasons:

- Configuration of hardware-related parameters such as disk partitioning is not possible on a running system where the disks contain live data.
- Configuration of very low level system software (such as basic networking) is difficult because the machine normally needs the network to be available before it can access the configuration database.

However, the "rotting" of static systems and the difficulty of identifying the configuration state of a particular machine can lead to many problems which make a dynamic system attractive.

Many vendors are now moving towards dynamic configuration systems based on object-oriented technology. The Tivoli "Management Environment", for example, is an object-oriented product which is available on several platforms. This provides a central configuration database and a "framework" into which objects can be slotted to control the various subsystems in a uniform way. Hopefully, standards will develop, and become adopted, so that multiple vendors (and users) can construct objects which inter-operate across heterogeneous systems. Although this provides the most promising future direction for system configuration, most vendors do not currently supply such software as part of their standard operating system package, and current implementations may be too expensive and/or inflexible for many sites.

### A Simple Dynamic Implementation

The Computer Science Department at Edinburgh University runs a network of 300-400 workstations with about 2000 users. System administration tools from the department are often adopted on a wider scale throughout the University. At present, these machines are mostly Suns (currently being upgraded to Solaris 2) and X terminals, but the ability to integrate systems from different vendors is considered very important and DEC, HP and SGI systems have all previously been integrated into the network. Particularly within research groups, such as the LFCS<sup>3</sup>, systems change rapidly and machine configurations are very diverse, so it is important to have a sufficiently flexible infrastructure to support this type of environment.

The *lcfg* ("local configuration") system [7] now being used in the Computer Science Department is a mainly dynamic system with a small amount of static configuration for the hardware and low-level

<sup>3</sup>Laboratory for Foundations of Computer Science.

parameters. All information that is necessary to distinguish one machine from another is contained in the central database and every machine can be rebuilt or duplicated using just the information from the database together with the generic system software<sup>4</sup>. Only Suns are currently being configured with `lcfg`, but it is intended that the system be portable, presenting a uniform interface to the configuration process across different platforms. The static part of the configuration which interfaces with Sun auto-install is the only part of the system which is expected to be significantly different on different platforms.

The static part of the configuration occurs when a machine is installed. Information is read from the database and used to construct auto-install configuration files determining the type of machine, the layout of the disks, the base software configuration, and other static parameters. When the machine reboots for the first time after an installation, a further script performs any remaining static configuration. This might include addition of clients or loading of additional software across the network. All machines can be installed entirely automatically, complete with all the necessary local customisation, simply by creating the database entries and booting the system from an install server.

Every time the machine boots, a script reads the configuration database to determine the *subsystems* that should be configured on that machine. This executes a script for each subsystem (for example, DNS or `xntp`) which consults the database for relevant parameters and dynamically configures the subsystem accordingly. New subsystems can therefore be incorporated into the configuration process simply by adding their names to the database entry for a specified machine. The dynamic configuration allows machines to be reconfigured very quickly to adapt to changing requirements, or work around failed hardware.

### The Configuration Database

The configuration scripts use common routines to consult the database for *resources* of the form

`host.subsystem.attribute = value`

In theory any database could be used to hold these resources and any mechanism could be used to distribute them to the client machines. A large relational database might be a useful tool for extracting information about machine configurations, and making complicated changes to groups of machines, but it is not strictly necessary and, at present, a simple flat file is used for each machine. The resources are distributed and supplied to the client machines using NIS[8]. NIS is not ideal for this purpose, since it involves propagation of the entire database every

time a single change is made, and all system software below the level of NIS must be statically configured. We hope to eventually develop a special protocol that operates at a lower level, but NIS is currently proving adequate as a resilient method of supplying machines with the necessary resources.

The information in the source files is deliberately of a very low level. As described later, the eventual aim is to generate this information automatically from a higher level description of the machine and its relationship to other machines in the network. At present, the files are edited by hand and passed through the C preprocessor which allows some degree of structure to be introduced, and machines with similar configurations to share common blocks of resources. A total of about 400 different resources are available for configuration of various different subsystems, but many of these will nearly always be used in their default values and a typical large server requires about 70-100 resources to fully describe the configuration. Clients usually require about half this number, and the use of the C preprocessor reduces the configuration description even further (some examples are given in the appendix).

Independent processes can very easily extract information from the database and one important application of this is to validate the consistency of the resources. A simple Perl script scans the resources for a specified machine and performs various consistency checks; the script is continually being extended to identify the most common configuration errors and this allows many problems to be detected before the machine installation has started. Since information is available on all machines, inter-machine problems can be located that might not normally be detected until a much later stage. In particular, it is possible to check before removing a machine from the network, that all dependencies on that machine have been removed. Not all of these dependencies are immediately obvious; for example, every ethernet segment must include a host supplying `bootparam` service, and removing the last `bootparam` server from an ethernet segment should cause a warning to be generated. Such checks can be used to identify weak points in the network by answering questions, such as "what happens if a particular server fails".

Some of the resources are purely informational and are used for administrative purposes (for example, the owner and location of the machine). One interesting application is an experimental *World Wide Web* service which makes information on all machines available over the World Wide Web by automatically querying the database when the page for a particular machine is accessed<sup>5</sup>. The

<sup>4</sup>Obviously backups of any user data are also required.

<sup>5</sup><http://www.dcs.ed.ac.uk/cgi-bin/hosts/INDEX>

information in the database allows hyper-text links to be generated between clients and their servers, and between personal workstations and the home pages of their owners.

### The Configurable Subsystems

Each configurable subsystem on a machine (for example, a printer) is a member of a particular *class* and the configuration for all subsystems in a class is performed by the same *class script*. All the class scripts share a number of common routines and are written in a stylised manner; this allows new classes with simple configuration requirements to be added very easily. A single subsystem called *boot* starts when the system boots. The resource *boot.services* is consulted to determine all the other subsystems that should be configured at boot time and the appropriate class scripts are executed. Provision is also made to execute these scripts manually, or at regular intervals (from *cron*).

There are currently about 30 different classes implemented, of which the following is selection:

- auth** configures all the authorisation of access to the machine. This controls, for example, the groups of users that are permitted to log in, and the machines to be included in *hosts.equiv* file.
- amd** controls the amd automounter, specifying the *cluster* that is to be used and hence determining the servers from which the various file-systems will be mounted.
- dns** controls the type of DNS service to be provided and (where appropriate) specifies the servers to be used.
- www** controls the World Wide Web server.
- xdm** controls the xdm subsystem specifying which X terminals are to be managed and configuring some of the parameters of the login session. A separate subsystem controls the font server.
- inet** controls the services that are managed by *inetd*, including the access control which is managed by the *tcpd* wrapper program.

The above subsystems run only when the machine boots, and any change in the database resources is not reflected in the corresponding subsystem until the machine is rebooted (or the subsystem is manually restarted). These are mostly one-off configurations (such as *auth*) or daemons which start once and run continuously (such as *www* or *xdm*). Some subsystems need to be run at regular intervals (for example, backups) and the *boot* subsystem can arrange to schedule these to run from *cron*. In particular, a group of processes runs every night to perform any necessary updates to the local file-systems:

**updatefile** uses *lfsu*[9] to update the local file-systems with any changes that have been made to the master copies of locally maintained software. The configuration of this subsystem determines the software packages that are to be carried by the machine.

**patch** applies any new systems patches that have been installed which are relevant to the machine.

**update** makes any necessary modifications to files in the root file-system to track the latest static configuration.

Most class scripts also accept additional arguments to stop and restart the subsystem, and to display logging and status information. A client program called *om*, and its associated daemon *omd*, provide a way to execute these additional *methods* remotely, including an authorisation scheme with access control based on the user, the host, the subsystem, and the method. This allows users to be given permission, for example, to stop and restart certain daemons running on their personal workstation. One possibility is that *om* will be extended to understand *netgroups* of machines, allowing subsystems to be easily restarted on a whole cluster of machines with a single command.

### High Level Configuration

One of the most important aspects of machine configuration is to specify the role of a machine within the network. This includes the relationship between a client and the servers which supply various different services. Typically, these will include file services of various types (home directories, program binaries), name service (DNS), time synchronisation (*xntp*), font service and others. If a client and server are configured independently, then there is no guarantee that the configurations are compatible; for example, a client can quite easily be configured to expect file service from a machine which is not exporting the required files, or even from a machine that does not exist! Even within a single machine, there are similar dependencies offering scope for errors when different subsystems are configured using different methods; for example, if a particular machine is to run a World Wide Web server, then the appropriate software must be available on the machine.

Using a common source of configuration information allows most of these dependencies to be checked automatically. However, the low level nature of the raw configuration resources means that production of configuration files is awkward and error prone. Ideally, we would like to describe the relationship between machines at a much higher level and have the low level configuration information generated automatically. For example:

- Machine A is the name server for the research group.
- Machine B is a member of the research group.
- Machine C is a member of the research group.

From the above specification, it is possible to generate all the necessary low level configuration information to load the name-server software, and start the name-server subsystem, on machine A, and configure the other machines to act as clients of this machine. An error (or at least a warning) would be expected for any machines which did not have a name-server.

The simple example given above can be accomplished quite easily, using features of the C preprocessor, with the existing implementation. Changing machine A to some other machine should cause the software and the daemon to be transferred to the other machine, and clients to change their `resolv.conf` files to point to the new server.

In addition to the essential rules, like the name-server example above, it is also very useful to be able to specify policy rules in a similarly explicit manner. For example:

- Students are not allowed to log in to personal workstations of staff members.
- File-servers which are updating local file-systems during the night should do so at different times to avoid network congestion.

Such policy rules are frequently contravened in practice because they are not critical to the operation of the system and mistakes can easily go unnoticed. Using the rules to actually generate the machine configuration guarantees that they will be enforced.

As the rules and their interactions become more complex, the need for a special-purpose *configuration language* to replace the C preprocessor quickly becomes apparent. Designing such a language [10] is not easy for several reasons; it must be able to express high-level rules in a clear, explicit way, but be capable of generating low level configuration information from these rules. Since the configuration subsystems must be extensible, the language itself must be extensible so that new rules can be added to control new subsystems, or new features of existing subsystems. Possible designs for such a language are currently under investigation.

### Conclusions & Further Work

The use of a dynamic configuration system storing parameters in a central database has been a big improvement over the previous static system. In particular:

- The ease with which configurations can be changed, and machines can be completely rebuilt, means that machine configurations do not "rot" and are always up-to-date.
- New subsystems can easily be introduced and configured onto existing machines without

interfering with other subsystems on the machine.

- The ability to validate and examine explicit machine configurations from the database has reduced the number of errors that are caused, for example, by forgetting some dependency when removing a server.
- Since the machines automatically reflect the configuration in the database, it is possible to have some confidence that policies specified in configuration rules are actually being enforced on the machines. This provides an improvement, for example, in security.

Disadvantages include the longer time required to boot a machine and the difficulty of manually creating correct low-level configuration information.

The ability to specify configurations and policies at a much higher level is a very useful facility. The best way in which to implement and exploit this possibility is an area for further investigation. In the short term, incorporation of further subsystems, porting to other platforms, and improvements to the mechanism for storing and distributing the resources are likely areas of future work.

### Availability

Copies of this paper and associated technical reports are available via WWW from <http://www.dcs.ed.ac.uk/staff/paul> or [pub/paul/papers](ftp://pub/paul/papers) on [ftp.dcs.ed.ac.uk](ftp://ftp.dcs.ed.ac.uk) (ftp).

### Acknowledgements

Thanks to all the systems staff of the Computer Science Department for long discussions on the design of the configuration system and for suffering all the machines with broken configurations during the development and testing.

### Author Information

Paul Anderson is a graduate in pure mathematics. He has taught computer science and managed software development before becoming involved in systems administration. He is currently employed as Systems Development Manager with the Laboratory for Foundations of Computer Science, where he is responsible for the research laboratory's network. He is also working with other system managers to develop the computing facilities within the department and the University. Paul can be reached by mail at:

The Laboratory for Foundations of Computer Science  
Department of Computer Science  
University of Edinburgh  
King's Buildings  
Edinburgh EH8 3JZ  
U.K.

His email address is: [paul@dcsc.ed.ac.uk](mailto:paul@dcsc.ed.ac.uk).

## References

1. Sun Microsystems, "Automatic installation," in *Solaris 2.3 system configuration and installation guide*, 1993.
2. George M Jones and Steven M Romig, "Cloning Customized Hosts (or Customizing Cloned Hosts)," *Proceedings of the LISA V Conference*, pp. 233-237, Usenix, 1991.
3. Jennifer G Steiner and Danial E Geer, *Network services in the Athena environment*, Project Athena, Massachusetts Institute of Technology, Cambridge, MA 02139.
4. Elizabeth Zwicky, "Typecast: beyond cloned hosts," *Proceedings of the LISA VI Conference*, pp. 73-78, Usenix, 1992.
5. Mark Rosenstein and Ezra Peisach, "Mkserv - Workstation customization and privatization," *Proceedings of the LISA VI Conference*, pp. 89-95, Usenix, 1992.
6. Rick Dipper, "Management information and decision support tools for Unix systems administration.," *Proceedings of UKUUG/SUG Conference*, pp. 143-153, UKUUG, 1993.
7. Paul Anderson, "Local system configuration for syssies," CS-TN-38, Department of Computer Science, University of Edinburgh, Edinburgh, August 1991. Available by anon ftp as file pub/paul/papers/tn38.ps from site ftp.dcs.ed.ac.uk.
8. Sun Microsystems, "The Network Information Service," in *System and network administration*, pp. 469-511, Sun Microsystems, 1990.
9. Paul Anderson, "Managing program binaries in a heterogeneous UNIX network," *Proceedings of LISA V Conference*, pp. 1-9, Usenix, 1991.
10. Bent Hagemark and Kenneth Zadeck, "Site - a Language and System for Configuring Many Computers as One Computing Site," *Proceedings of the LISA III Conference*, pp. 1-13, Usenix, 1989.

## Appendix 1: Configuration for a Simple Server

```

/*****
Staffa
*****/

#include <lfcs.h>

/* Resources for information only */

info.type          server
info.location      the machine halls
info.make          Sun
info.model         10/40
info.owner         LFCS
info.memory        16 16 16
info.sno           411m1238
info.hostid        727099f2
info.disks         internal wren
info.disktype_internal SUN1.05 cyl 2036 alt 2 hd 14 sec 72
info.disksize_internal 1Gb
info.diskdev_internal c0t3d0
info.disktype_wren  CDC Wren VII 94601-12G cyl 1929 alt 2 hd 15 sec 68
info.disksize_wren  1Gb
info.diskdev_wren   clt1d0

/* Statically configured resources */

install.system_type server
install.arch         sun4m
install.client_arch  sun4c sun4m
install.local        B_INSTALL_CONFIG
install.interfaces   le0 qe0
install.hostname_le0 HOSTNAME
install.hostname_qe0 HOSTNAME-j
install.updatelf      true
install.install_server true
install.filesystems   root swap var usr export local
install.fs_root       c0t3d0s0 32 /
install.fs_swap       c0t3d0s1 64 swap
install.fs_var        c0t3d0s3 64 /var
install.fs_usr        c0t3d0s4 auto /usr
install.fs_install    c0t3d0s7 350 /export/install
install.fs_export     c0t3d0s5 free /export
install.fs_local      clt2d0s2 all /disk/local

/* Dynamically configured resources */

auth.rootpwd        LFCS_SERVER_PASSWD
auth.users          LFCS_SERVER_USERS
auth.equiv          LFCS_EQUIV
auth.rhosts         LFCS_RHOSTS
amd.cluster         HOSTNAME.dcs.ed.ac.uk
dns.type            server
yp.type             slave
yp.servers          HOSTNAME
boot.services       SERVER_SERVICES
boot.run            SERVER_RUN
cron.objects        boot
cron.run_boot       0 0 * * *
updatelf.fs         local
updatelf.fs_local   sun4-51 share
updatelf.netgroups  delete copy
updatelf.action_copy copy

```

```

updatelf.action_delete  delete
nfs.exports             local
nfs.fs_local            /disk/local
nfs.options_local       -o ro=machines

```

## Appendix 2: Configuration for a Simple Diskless Client

```

/*****
 * Gasker
 *****/

#include <lfcs.h>

/* Resources for information only */

info.type                private
info.owner               paul
info.location            1612
info.make                Sun
info.model               Classic
info.sno                 302U4308
info.hostid              8001d534

/* Statically configured resources */

install.system_type      client
install.arch             sun4c
install.interfaces       le0
install.hostname_le0     HOSTNAME
install.root              B_SERVER:/export/root/HOSTNAME
install.swap              B_SERVER:/export/swap/HOSTNAME

/* Dynamically configured resources */

mail.root                paul
auth.rootpwd             LFCS_CLIENT_PASSWD
auth.users               LFCS_CLIENT_USERS
auth.equiv               LFCS_EQUIV
auth.rhosts              LFCS_RHOSTS
amd.cluster              B_SERVER.dcs.ed.ac.uk
dns.servers              B_SERVER
cron.objects             boot
cron.run_boot            0 4 * * *

```

# OMNICONF – Making OS Upgrades and Disk Crash Recovery Easier

*Imazu Hideyo – Matsushita Electric*

## ABSTRACT

OS upgrades are a headache because after installing a new OS, many files and directories need to be modified or created by hand, to restore the host's previous (pre-upgrade) configuration.

On the other hand, saving entire / and /usr file systems for crash recovery is redundant because most files are unchanged, and copies exist on distribution media. In addition, restoring from backups after a disk crash is not as easy as an OS installation from distribution media because OS installation software does not necessarily include utilities to aid in doing so.

Difficulties in performing OS upgrades and disk crash recoveries are dramatically reduced if a complete set of "changes" (a set of changes is called a "configuration" in this paper) which have occurred throughout / and /usr can be observed and saved. "Change" means: 1) addition and deletion of files and directories; 2) modification of the content and status of files and directories. Dealing with changes is non-trivial because conventional commands such as tar, cpio, and dump cannot handle deletion and cannot alter the permissions of a file without restoring its contents.

If configurations can be stored under a single directory, OS upgrades become easier because the configuration can be restored by a simple operation after the upgrade. Instead of saving all files in / and /usr, one only needs to save changes to those file systems. One can easily perceive what the entire configuration is and modify merely a part of it.

In this paper, the author introduces a tool called "OMNICONF", which stores and restores "configurations" to and from a specified directory. OMNICONF is implemented in 2400 lines of Perl[1] code, under the concept shown above.

## Motivation

Computers must be configured to be used practically. With UNIX, configuration means (for the most part) modifying or creating files and directories in the /, /usr, and /var file systems.

OS upgrades can be a nightmare if the administrator has changed and created many files in /etc, /usr/lib, /usr/etc, /usr/sbin, etc. In addition, he may have made directories for mount points, device files for additional devices, and symbolic links in the course of system configuration. He might change the mode, owner, or group of files and directories. A typical UNIX system has between 20 and 100 modified or created files and directories in the /, /usr, and /var file systems. After an OS upgrade, these operation need to be done again, usually by hand. In most cases, there is no complete list of such changes. Ordinary administrators, therefore, must rely solely on remembering these past operations and must repeat them precisely after an OS upgrade.

As well as OS upgrades, recovering from a disk crash can also be difficult. Information on how to reinstall the system from backup may be hard to find or such an operation may be difficult to accomplish correctly. In the past, there was no such thing as an

automated OS installation program, thus administrators did installations manually: set disk partitions, make file systems with the "newfs" or "mkfs" command, read OS tapes with the "tar" or "restore" command, and so on. Today, convenient installation programs do this work for the administrator. Therefore, information and tools for manual installation become hard to find today and are sometimes buggy, creating a complicated situation. For example, can you correctly install a boot block by hand when BSD/386 is in the second FDISK partition of an PC-AT compatible machine? Since recovery from backup is similar to performing a manual installation, it is harder to do correctly today.

## Concept

To deal with problems described above, the author has developed the OMNICONF system. This section describes the design concept of OMNICONF.

Assume that the changes to files and directories since the original OS installation has been stored. The author defines the set of changes (or "configuration" hereafter) as the following items:

1. The contents of modified and created files
2. A list of removed files and directories

If a configuration can be stored and restored, an OS upgrade is much easier: install the new version of the OS, then apply the configuration. In reality, the administrator may also need to modify the configuration if he applies it to a new version of the OS, because the format of certain file may have changed, or some files made redundant. However, the stored configuration aids greatly in the upgrade task because it is a complete list of changes. There is no longer a need to remember changes and implement them manually.

The benefit of the stored configuration is more dramatic with disk crash recovery. All the administrator needs to do is to reinstall the OS normally and then apply the configuration. There is no hassle for OS installation from backup media, and the time needed for recovery is reduced. (because in many cases, OS software is distributed with CD-ROM and installation from CD-ROM is faster than from tape, which is a typical backup medium). Requirements

Under this concept, what is required for OMNICONF to be a reasonable aid for system administration?

First, the system should handle any type of file: ordinary files, directories, symbolic links, and device files.

Second, modified files must be stored hierarchically. Assuming modified files are stored under the /config directory, a modified /etc/sendmail.cf would be stored as /config/etc/sendmail.cf and a modified /usr/lib/sendmail would be stored as /config/usr/lib/sendmail. If a configuration is stored as monolithic data, it becomes much more difficult to manipulate only part of a configuration before applying it to an upgraded OS.

OMNICONF can deal with changes in file type. For example, /tmp is originally a directory, but it may be changed by the administrator to become a symbolic link pointing to a new location, e.g., /var/root.tmp||.

Administrators may change the mode, ownership, or group (the combination of this information will be hereafter referred to as an "attribute") of files and directories while keeping their contents intact. Attribute changes should be observed and saved without storing the content of a file. If the contents of a file is stored in such a case, the contents may be needlessly or harmfully applied during an OS upgrade.

System files may be removed for administration purpose. For example, to disable the routed daemon without modifying /etc/rc.local, one may instead remove /usr/etc/in.routed. The removal of files and directories must be observed.

When restoring the configuration, original files should not be overwritten, but should be stored elsewhere, since one may need to refer to the original

versions of files such as sendmail.cf, inetd.conf, syslog.conf, etc. at a later date.

For some files, a certain command needs to be invoked after the file is modified. For example, the newaliases command should be invoked after the /etc/aliases file is modified. This sort of binding should be handled.

Even if a program that performs the above tasks works correctly, no one will want to use it if they need to maintain the complete list of changed files and directories by hand. Such a list should be generated automatically and maintained dynamically.

### Conventional Commands

Commands such as dump, tar, and cpio can store only files newer than a certain date and time, and these conventional commands lack some essential functionality. For example, they overwrite files and do not save original files, they fail to create a symbolic link if a directory with the same path name already exists, they cannot restore attributes without restoring content, and they do not correctly handle file deletion.

### Features

Now that the requirements have been stated, the author will describe the features of OMNICONF system.

#### List Changes

The key issue in storing a configuration is to list all modified/created/removed files and directories automatically. For this purpose, OMNICONF uses a list of all files and directories, which is created by a special command when the OS is installed or a machine is unpacked. This list is called the original "profile" of the OS. Here is an example of a portion of a profile file:

```
/:40755:0:0:767587342
/.cshrc:100644:0:10:711924842
/.login:100644:0:10:711924842
/.profile:100644:0:10:711924842
/.rhosts:100644:0:10:711924842
/bin:120777:0:0:767585902:usr/bin
/boot:100444:0:3:767586993
```

Each line denotes a file or directory and consists of colon-separated fields. The first field is the path name of the file. The second through fifth fields are pieces of data that are returned by the stat system call. The second field is the mode of the file (in octal), the third field is the UID of the file's owner, the fourth field is the GID, and the fifth field is mtime (when the file was last modified). Symbolic links have a sixth field, which is the contents of the link. The profile file of a SunOS 4.1.3 installation without any configuration contains 8,871 lines and is 440,867 bytes in size. The size varies, depending on which software sets are installed.

The original profile of an OS is stored in `/etc/omniconf/profile`. When a configuration is stored, the current profile and the original profile are compared. OMNICONF determines whether the contents of a file has been changed by comparing mtime's. Change of attribute is determined in a similar fashion. A file that is not listed in the original profile but is listed in the current one is determined to have been created. Deleted files can be determined similarly.

What portion of the entire file space should be handled? The administrator should specify this information in order for OMNICONF to work correctly. First, the file systems to be managed should be specified. Second, files and directories to be excluded should also be specified because saving certain files can be redundant or even harmful. For example, files that store system status, such as `/etc/utmp`, `/etc/mstab`, etc., should not be manipulated by OMNICONF. Files generated from other files such as `/etc/aliases.dir`, also should not be saved.

OMNICONF refers to `/etc/omniconf/area` to determine what portion of an entire file space is handled. Here is an example of area file for SunOS 4.1.3.

```
filesystem:
/
/usr
/var
excluded:
.pid$
.lock$
/etc/aliases.dir
/etc/aliases.pag
/etc/dumpdates
/etc/ld.so.cache
/etc/mstab
/etc/psdatabase
/etc/state
/etc/ttys
/etc/utmp
/dev/console
/dev/null
~/dev/tty
~/dev/pty
/var/adm
/var/log
/var/tmp
```

The file consists of two portions: the file system portion and excluded portion. The file system portion is straight forward. The excluded portion contains path names and regular expressions. Entries that begin with a slash are taken as path names and otherwise are regular expressions. Files and directories that match excluded entries are ignored. A path name entry in the excluded portion matches the path itself and any descendant files and directories (children).

## Preserve Originals

OMNICONF assumes that the original version of a file whose content may be modified is saved as `*.orig`. For example, the original `/etc/sendmail.cf` can be saved as `/etc/sendmail.cf.orig`. But one doesn't have to preserve original files if they are not needed.

When the administrator want to disable the routed daemon by deleting `/usr/lib/in.routed`, he has two alternatives: rename the file or unlink the file. OMNICONF recognizes deleted files and directories by renaming them `*.orig`. For example, an administrator may rename `/usr/etc/in.routed` to `/usr/etc/in.routed.orig`. In this case, OMNICONF considers `/usr/etc/in.routed` as having been removed. OMNICONF also handles unlinked files and directories.

Modified or removed files and directories whose originals are preserved can be reverted to their original versions by a certain OMNICONF operation.

## Store Configuration

A configuration is stored under the directory (referred to hereafter as the "repository") specified by `/etc/omniconf/reposit`. Assume that the repository is `/config` for example. The repository contains one directory named 'cont' (in this case `/config/cont`), and two files named 'remove' and 'chstat.'

Files and directories that are modified or created are stored hierarchically under the cont directory preserving mode, owner, group, and mtime. For instance, `/etc/sendmail.cf` is stored as `/config/cont/etc/sendmail.cf`.

In the `/config/remove` file, removed files and directories are listed line by line. An example of remove file is as follows:

```
!/etc/hosts.equiv
/usr/etc/in.routed
```

Unlinked files and directories are prepended with a '!.' Files and directories renamed `*.orig` and not unlinked are listed as just their path names.

`/etc/omniconf/chstat` contains attributes of files and directories whose attributes have changed but whose contents have not. Here is an example of chstat file:

```
/var/spool/uucppublic:644:4:8
/var/tmp:1777:3:10
```

Each line consists of a path name, permissions, UID of owner, and GID of group. Although all mode bits are stored in profile files, only permission bits, which are mode bits masked by 0777 (octal), are stored in the chstat file.

## Command Binding

The binding of a file to a command is specified by `/etc/omniconf/exec`. Here is an example:

```

/etc/aliases
newaliases
/etc/named.boot
if [ -f /etc/named.boot ]; then \
    named.restart \
else \
    kill 'cat /var/run/named.pid' \
fi
/vmunix
fastboot

```

File names begin in column one, and bound commands have spaces or tabs at the beginning of lines. Much like the “make” command, command bindings are evaluated by /bin/sh. Order in the exec file is significant: entries in the file are examined in the order in which they appear.

### Restore Configuration

The process of restoring a saved configuration consists of following steps.

1. Remove files and directories according to the remove file.
2. Change the attribute of files and directories listed in the chstat file.
3. Rename files whose new version exists in the repository. The original files are renamed as \*.orig.
4. Copy files and directories with the cpio command.
5. Examine the exec file and invoke commands as specified.

### Elements of OMNICONF

The current OMNICONF system is written in about 2400 lines of Perl code. It consists of following commands:

- mkoprof (Make OMNICONF profile), which is used to create an original profile in /etc/omniconf/profile when an OS is installed.
- putconf, which calculates difference between the current profile and the original profile, then stores the configuration under the repository directory.
- getconf, which reads the repository and restores the saved configuration.

These commands should run without a Perl interpreter since there may be no Perl interpreter available when they are invoked. Using the undump feature of Perl, the commands are made into pure executables.

Putconf and getconf use GNU cpio to write and read the repository. Standard cpio is inadequate since it cannot handle the Berkeley Fast File System properly.

### Real Operations

After one has installed an OS or unpacked a UNIX machine, copy the mkoprof command to local disk (such as to /tmp) and invoke it to create

/etc/omniconf/profile. On a SPARCstation 2 with SunOS 4.1.3, mkoprof takes about 2 minutes to complete, and the resulting file consists of 8,871 lines (440,867 bytes).

After the machine has been configured, install putconf, then prepare a repository directory, and create /etc/omniconf/area and /etc/omniconf/reposit.

When the administrator wants to store a configuration, he invokes putconf. Putconf takes about 2 minutes to calculate the configuration in the example above, then putconf takes up to an additional 30 seconds to store data. A configuration typically amounts to between one and several megabytes. The administrator may want to save the file hierarchy under the reposit directory onto backup media. Compared to an entire OS area of between 50 and 150 megabytes in size, saving only a configuration consumes much less storage (and thus backup time).

When the system disk of a machine crashes and its OS is reinstalled, the administrator must restore the configuration previously saved onto backup media. In some cases, the repository may not have been affected by the failure. In that case, simply copy getconf to local disk and invoke getconf, specifying the repository directory as an argument.

An upgrade operation is not as simple as disk crash recovery. The administrator has to examine files and directories stored as the configuration of the previous version, and make some changes. For example, a kernel image (/vmunix, /unix, /bsd) should be removed. Then he can install the configuration by invoking getconf. OMNICONF helps OS upgrade procedures mainly for minor OS upgrades such as from SunOS4.1 to SunOS4.1.3. and from BSD/386 1.0 to BSD/386 1.1. Major OS changes such as from SunOS4.1 to Solaris 2.3 cannot be simplified with OMNICONF, because formats and locations of configuration files may be changed.

### Repository on a Different Machine

#### The Feature

The repository can be placed on a different machine, which means OMNICONF can save and restore a configuration of a machine to and from another machine. In this case, /etc/omniconf/reposit contains the name of the machine that has its repository. Another perl script named “omniconfsrv” should be installed on the machine that will store the repository. When a configuration is stored on another machine, putconf invokes omniconfsrv via rsh instead of storing configuration files with cpio. Similarly, getconf will restore a configuration by invoking omniconfsrv via rsh.

#### Remote Configuration

Files and directories in a repository can be manipulated with file manipulation commands such as vi, chmod, chown, mkdir, etc. Getconf will

transfer the manipulation to the machine whose repository was manipulated. This allows one to manage the configuration of a machine by manipulating its repository. For example, assume that the machine "aries" has its repository in /config/aries on the machine "taurus". If the administrator creates the directory /config/aries/mnt1 on taurus, then invokes getconf on aries, the directory /mnt1 is made on aries. Since OMNICONF handles all aspects of a configuration, anything concerning configuration can be done remotely.

#### Comparison with Other Systems

The Track[2] System propagates a configuration to uniformly configured machines on a large scale. If the same repository is shared among several machines, OMNICONF can also do such a thing. But since the repository machine's load may be rather high, it may be inadequate to use OMNICONF on a large scale under the current implementation.

#### Conclusions

OMNICONF is very useful for crash recovery and OS upgrades, but it has a few shortcomings: the contents of /etc/omniconf/area should be determined by a "cut and try" manner, and the putconf command takes a relatively long time to execute, but this system is worth using it. Using OMNICONF, an administrator can concentrate configurations of several machines on a single repository machine. He can then manipulate the configuration of any machine by making changes on the repository machine.

The author is using OMNICONF on SunOS4.1 and BSD/386. By the virtue of OMNICONF, the configuration procedure of a machine running BSD/386 is 1.1 completed almost instantly.

OMNICONF is still a premature system. It should be used in more systems and by more administrators to develop into a reliable and capable tool.

#### Availability

OMNICONF is available from Information and Communications Lab., Matsushita Electric under a license agreement. Please contact the author of this paper for details.

#### Acknowledgements

The author would like to thank Yoshida Jun, manager, and Kushiki Yoshiaki, director of our lab for giving me the chance to develop OMNICONF. Special thanks are given to Utashiro Kazumasa of SRA for valuable suggestions on this scheme. The author also thank Ohtsu Takashi of our lab, who has patiently brushed up the English in this paper. Finally, the author really appreciates Kennedy

LEMKE of Panasonic Technology Inc. for his proof reading.

#### Author Information

Imazu Hideyo (Imazu is his family name) earned a Masters degree in Computer Science from the Tokyo Institute of Technology in 1988. Since then, he has been working for Information and Communications Lab., Matsushita Electric as a network administrator. He can be reached via snail mail at Information and Communications Lab., Matsushita Electric, Osaka-hu Kadoma-si Kadoma 1006, Japan or electronically at himazu@isl.mei.co.jp.

#### References

- [1] Larry Wall, Landal L. Schwartz, *Programming Perl*, O'Reilly and Associates, 1990.
- [2] Daniel Nachbar, *When Network File Systems Aren't Enough: Automatic Software Distribution Revisited*, Proceedings of the Summer USENIX, Atlanta, GA., June 16-18, 1986.



# Automated Upgrades in a Lab Environment

Paul Riddle – University of Maryland, Baltimore County

## ABSTRACT

Back in the late 80s and early 90s, when disk drives were expensive, it was more economical to buy one server and configure it with enough disk space to support several "diskless" workstations. Now that disks are cheaper, most workstations now come with internal disks which contain an entire bootable operating system.

Most vendors provide ways of automatically upgrading multiple "diskless" workstations; unfortunately, the same is not true for "diskfull" configurations. Upgrading "diskfull" workstations typically involves either a lot of manpower or a lot of tedious, repetitive work. In any moderate to large sized network, something needs to be done to automate the upgrade process.

This paper describes a scheme which we use to upgrade our various networks of Silicon Graphics workstations. Interestingly, it relies on the same technology that allows "diskless" workstations to boot over the network.

## Introduction

Our upgrade scheme works using diskless booting. Each workstation boots over the network from another workstation, which we designate as an "upgrade server." Once booted, the workstation runs an upgrade script (written in *Perl* [1]) which partitions its system disk, creates filesystems, installs an operating system distribution, and then installs customized system files. When finished, the workstation reboots from its system disk. This scheme allows for unattended system upgrades and has proven to be quite flexible; we have used it to upgrade two separate networks of SGI Indigos from Irix 4.0.5 to Irix 5.2.

## What We Were Looking For In An Upgrade Scheme

### Automation

The upgrade procedure should not require that we physically visit every workstation. This is a problem in our environment where many workstations are located in private offices to which we don't have easy access. Visiting each machine also requires a lot of manpower and can be error-prone; operator errors can lead to machines being upgraded incompletely, improperly, or not at all.

### Flexibility

An upgrade scheme should be able to deal gracefully with different sized system disks, different models of a vendor's workstations, etc. It should be able to repartition and create filesystems on the machine's local disk, if necessary.

### Reliability

The upgrade procedure should be reliable. It should never leave a machine in a partially-upgraded state. If an upgrade is interrupted or otherwise fails, it should pick up where it left off, or start over the next time the machine is rebooted. It should have some way of notifying the system administrator when an upgrade fails or completes successfully.

### Speed and Convenience

Upgrading should be reasonably fast and should not require a lot of downtime. Alternatively, it should be automated to the point where it can be done overnight, when there is less demand for workstations and network bandwidth.

## Our Environment

The University of Maryland, Baltimore County is one of the largest educational installations of Silicon Graphics (SGI) equipment in the country. There are approximately 200 SGI workstations on campus, spread out over about 8 different administrative domains and 10 subnets. The abundance of SGIs required us to come up with some way of keeping them up-to-date with the latest release of Irix (SGI's flavor of UNIX). We chose two different workstation networks to use as "Guinea Pigs" for testing our upgrade scheme.

For one upgrade environment, we used three student labs consisting of a total of about 90 SGI Indigos, some with entry level (RPC) graphics, and others with extended (XS24 or Elan) graphics. Each workstation has a 420-megabyte internal system disk. The workstations are spread over two different subnets.

A second upgrade environment consisted of about 30 SGI Indigos, mainly with low-end graphics, and 10 SGI Indy systems. Some of these machines have 420-megabyte system disks and others have 1-gigabyte system disks. All are on the same subnet.

For both environments, the task was to upgrade from some revision of Irix 4.0.5 (4.0.5F in some cases and 4.0.5H in others) to Irix 5.2.

### Alternatives To Our Approach

We evaluated several other methods of upgrading before choosing to implement one based on diskless booting. Each of these has its advantages, but fails to meet our requirements in one or more ways.

#### Upgrading Systems Individually

The most obvious and straightforward upgrade strategy is simply to upgrade systems manually, one at a time. We discarded this idea quickly because it was too time consuming. It also requires physically visiting each workstation. Additionally, manually upgrading a workstation is a tedious process which involves many steps. When many workstations are upgraded in this way, it can lead to subtle differences and inconsistencies between systems.

#### Manual Disk "Cloning"

A faster method is to upgrade manually one of each different type of system, and then upgrade the rest of the machines using a sector-by-sector disk copy. This is much faster and more reliable than upgrading individually, but still requires physically visiting each and every machine. The disk cloning doesn't extend to systems with differing system disk geometries, either. For example, you can't clone a 1-gigabyte system disk onto a 420-megabyte system disk; it just doesn't work. Operator error also creeps into the picture; although you're less likely to end up with inconsistencies between systems, there is still a good chance that machines can be missed or otherwise improperly upgraded.

Although this method doesn't really meet our needs, we did use it for awhile because it is simple and straightforward. Trained student employees provided the manpower.

#### Upgrading Running Systems With *rdist*[2]

Still another approach was to use *rdist* or a similar tool to upgrade a running system[3]. This worked well for a minor OS revision, but was not capable of handling a major revision such as upgrading from Irix 4.0.5H to Irix 5.2.

#### Using Unused Swap Space For Upgrade Filesystem

Another method was to use unused swap space to create an upgrade filesystem. SGIs allow swap to be removed from a running system, so it was possible to dynamically delete enough swap to create room for the upgrade filesystem, boot from there, and upgrade the system disk over the network.

However, this approach is not 100% reliable, since there's a chance that adequate swap space may not be available at upgrade time. Also, this approach doesn't allow for repartitioning the system disk during the upgrade, since part of the disk is in use as the upgrade filesystem.

### Our Solution

In designing an upgrade scheme, we worked to come up with a solution that satisfied all of our criteria: automation, flexibility, reliability, and speed. An important requirement was to avoid having to visit each workstation individually. This ruled out any solution involving disk "cloning" or upgrading individually from CD-ROM. We worked around this by having workstations copy the operating system over the network from a server.

In order to do this, the workstation needs to be booted to a state where its network interface is operational and its system disk is not being used. Enter diskless booting.

Diskless booting is an attractive solution because it allows for complete control of the system disk when performing the upgrade. The disk can be reformatted, repartitioned, mounted, unmounted, etc. at will. However, diskless booting is not without its problems. The booting protocol requires that the upgrade server be located on the same logical network as the client being upgraded. Many simultaneous upgrades can place an undesirable load on the network. The next section describes how we worked around the former problem. For the latter problem, we place limits on the number of simultaneous upgrades at the expense of time.

### The Upgrade Procedure

Doing upgrades is a three-step process. First, you need to configure each upgrade server to support diskless clients. Then, you must do a prototype installation for each different type of environment you are supporting. Finally, each workstation needs to be configured to boot from the upgrade server and then rebooted to start the upgrade process.

#### Configuring Servers For Diskless Booting

The first step in configuring the upgrade server is to build the diskless booting area. Let's assume that the hostname for the upgrade server is *sonata*. The upgrade area is rooted on *sonata* under */upgrade*.

The upgrade area contains everything that a workstation needs to boot diskless over the network and perform its upgrade procedure. A minimal number of OS files are necessary to support a diskless environment. All prototype and site-dependent distribution trees also live under the upgrade area.

Prototype distributions are located under */upgrade/proto*. Under recent releases of Irix, machines with different graphics boards and/or

processors require slightly different installations of the operating system. Each installation requires a separate prototype tree. For example, if your site has R4000 Indigos with entry (RPC) graphics and R3000 Indigos with Elan graphics, you would need two prototype distributions, which might be called */upgradel/proto/4krpc* and */upgradel/proto/3kelan*. (The names are arbitrary; you can choose whatever names you want.) Under these trees would be two prototype Irix installations, one for both machine architectures. Prototype distributions are either disk images or filesystem images generated by *dump*; the next section describes how to generate them.

We were able to work around the need for multiple prototype distributions by making several modifications to the default Irix distribution provided by SGI. This was done at the cost of a few extra megabytes of disk space on each system, which we decided was an acceptable tradeoff. The specific details of our modifications are beyond the scope of this paper, but we will make them available via FTP along with the rest of our upgrade tools.

Site distribution trees are located in */upgrade/dist*. Once the client has copied the appropriate prototype distribution, it uses *rdist* to copy selected site distribution trees. These trees contain system files which need to be modified from the defaults supplied by SGI, and any additional site-dependent files which need to live on the workstation's local disk.

The main purpose of site distribution trees is to separate customized files from standard files. This reduces the possibility that customized files will be lost when doing an upgrade.

The */upgrade* tree must be exported to all clients. Each client mounts */upgrade* as its root filesystem. To allow for multiple simultaneous upgrades, we export */upgrade* readonly and take pains to ensure that the clients do not try to write to it.

### Building Prototype Environments

To build prototype installations, we manually upgraded one of each type of workstation and then copied the resulting installation onto an external hard disk. Putting the distributions on an external disk allowed us to move them around from machine to machine, thereby enabling us to set up installation servers on different subnets. We found that a 1.6 gigabyte external drive was large enough to hold two separate prototype installations.

For networks with identically-sized system disks, we used *dd* [4] to copy the disk image over to the external drive. This is the fastest way to do things. Unfortunately, it doesn't work on networks where workstations have system disks with differing geometries. In this case, we used *dump*, [5] which is slower, but works on any disk regardless of geometry and partitioning. *Dump* also requires a separate prototype file for each filesystem on the

client's disk. For example, rather than a single disk image, we might have two separate *dump* images called */upgradel/proto/3kelan.root* and */upgradel/proto/3kelan.usr*.

Once we built all of the prototypes, we attached the external drive to the upgrade server and mounted it under */upgradel/proto*.

### The Upgrade Procedure

Workstations upgrade themselves using the following procedure. First, each client must be configured to boot diskless from its upgrade server. On Silicon Graphics boxes, this is done by setting two variables in non-volatile RAM (nvram) on each client:

```
client# nvram diskless 1
client# nvram bootfile \
    bootp()sonata:/usr/etc/boot/upgrade/unix
client# /etc/reboot
```

We did this for each of our clients using a simple shell script. Other methods include *rdist*, *cron*, etc.

When the workstation reboots, it loads the kernel image specified in nvram and mounts */upgrade* via NFS[6] from the upgrade server, *sonata*, as its root filesystem. It then starts up the init process, which in turn runs */etc/rc*.

*/etc/rc* begins by reading the "netaddr" variable from nvram, which contains the client's IP address. It looks up this value in the */etc/hosts* file to determine the system's hostname, and then configures the network interface.

Next, */etc/rc* execs the upgrade program, which is a Perl script. The script has four basic functions:

1. Repartition the local disk (optional). If desired, the existing partitions can be used.
2. Create new filesystems on the local disk.
3. Copy a prototype Irix distribution from the upgrade area to the local disk. As mentioned earlier, the prototype distributions are virgin Irix distributions installed directly from CDROM, with no modifications. This is done using *dd* or *dump*, invoked using a remote shell from the client to the upgrade server. A sample *dd* command would look something like

```
rsh sonata dd ibs=32768 obs=1450 \
    if=/upgrade/proto/3kelan | \
    dd ibs=1450 obs=32768 \
    of=/dev/rdisk/dks0d1vol
```

The blocking factor is determined by taking the MTU of SGI's ethernet interface and subtracting 50 bytes for TCP overhead. The output from the disk image is sent directly to the client's raw disk device.

4. Copy any number of site-specific distribution trees on top of the new OS distribution. This is where all customized system files are

installed. The copy is done by invoking *rdist* on the server via remote shell. For example:

```
rsh sonata rdist -c \
/upgrade/dist/3kelan client-hostname:/
```

Once this has finished, the upgrade script resets the nvram variables and reboots the system with the newly installed operating system. If any part of the upgrade is interrupted (due to someone turning off or resetting the machine, power failure, etc.), the upgrade procedure will start over when the system reboots.

### Performance Observations

We found that it took approximately 20 minutes to copy a 420-megabyte disk image over a lightly loaded ethernet using *dd*. Using *dump*, the procedure took about 30 minutes. By comparison, a direct sector-by-sector disk copy took around 10 minutes. Unfortunately, ethernet doesn't have quite the bandwidth required to upgrade more than one workstation at a time. We found that the most efficient way to get the upgrade done was to write a script that upgrades each workstation sequentially, and let it run overnight.

### Future Work

Currently, our scheme requires that the root, swap and usr partitions be allocated to specific partitions on the local disk. This works fine for just about any workstation. However, we would like to expand this to be a bit more flexible and support customized configurations.

Our scheme is also very dependent on NFS. We'd like to eliminate NFS from the picture (except where it is required for diskless booting) and switch to a different method of copying the prototype areas. FTP[7] appears to be a very attractive solution.

Maintaining separate prototype distributions can eat up a lot of disk space. We have a couple of ideas which would alleviate this problem. One is to use actual running systems as prototypes; however, this would require a different upgrade server for each individual machine type, which may be difficult to do in terms of network topology. Another solution, one which we may implement in the future, would be to have each client do a direct install from a CDROM server. Unfortunately, this tends to be a slow process, and also requires a front end (such as *expect*[8]) to drive the installation process. *Expect* scripts would need to be tailored for each different Irix release, which would be tedious and problematic.

The current method we use to initiate an upgrade is somewhat of a kludge. It would be nice to have a server/client type protocol which allows the admin to start upgrades remotely and monitor their progress.

### Author Information

Paul Riddle is a Systems Programmer with Academic Computing Services at the University of Maryland, Baltimore County (UMBC). He has been working at UMBC since 1989. When he graduated in 1992, he made the transition from underpaid student to full-time employee. Currently, Paul works with *sendmail* and DNS, and helps to keep the student labs running, among other things. Someday he hopes to become motivated enough to get a Master's degree, too. Reach him via U.S. Mail at The University of Maryland, Baltimore County; 5401 Wilkens Avenue; Baltimore, MD 21228. Reach him electronically at paulr@umbc.edu.

### Availability

We expect to have the final version of our upgrade software ready by September 1, 1994. It will be available via anonymous FTP from *ftp.umbc.edu* in the directory */pub/sgi/upgrade*.

### References

- [1] Wall, L., & Schwartz, R., *Programming Perl*, O'Reilly & Associates, Inc., 1990.
- [2] "rdist(1C) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
- [3] Manning, C., and Irvin, T., "Upgrading 150 Workstations in a Single Setting", *Proc. 7th Usenix Systems Administration Conference (LISA VII)*, 1993.
- [4] "dd(1M) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
- [5] "dump(1M) Manual Page," *IRIX Reference Manual*, Silicon Graphics, 1993.
- [6] "NFS Protocol Specification," *Networking on the Sun Workstation*, Sun Microsystems, 1986.
- [7] Postel, J. & Reynolds, J., "File Transfer Protocol (FTP)," *RFC 959*, Network Information Center, 1985.
- [8] Libes, D., "Using *expect* to Automate System Administration Tasks", *Proc. 4th Usenix Systems Administration Conference (LISA IV)*, 1990.

# Tenwen: The Re-engineering Of A Computing Environment

Rémy Evard – Northeastern University

## ABSTRACT

In the summer of 1992, the computing environment at the College of Computer Science of Northeastern University was completely dysfunctional. Among other things, the network was down over 25 percent of the time, the computers and software were badly misconfigured, the users were confused, and it was nearly impossible to administer. It was on the verge of collapse.

Now, two years later, the situation is entirely reversed. The network is up well over 99 percent of the time, the computers and software are easily managed, and the users are (for the most part) satisfied. Many people have stated that it is the nicest and most functional computing environment they've ever used.

This paper is an overview of the key changes that we made and the methodology that we used in bringing about this transformation. I examine the lessons we learned as well as the mistakes we made, and offer advice to others starting with a similar predicament.

## Introduction

The College of Computer Science at Northeastern University currently operates a network consisting of approximately 350 computers of various types. About 1200 people use the systems for education and computer science research. The Systems Group of the College, which consists of four full-time staff and a variable number of students, is responsible for the administration of the whole network.

We support several different types of UNIX workstations, as well as Macintoshes and PCs, all of which have hundreds of applications installed. The network as a whole has been up constantly for the last nine months. The user community is about as content as a group of 1200 people can reasonably be expected to be, and occasionally even compliment us on the way the computers and software run.

Two years ago, the situation was entirely different. The two main computing staff members, who had been primarily VAX-style operators, had just left the school. The facilities consisted of about 100 barely usable UNIX computers, several partially networked Macintoshes, a micro-VAX, and a convoluted network that was down more than it was up (or so it seemed to the users). By all reports, the environment was very nearly unusable.

The rest of this paper describes the process of moving from the dysfunctional system of two years ago to the current environment. I present a chronological overview in order to place the changes in perspective, and then summarize the key ideas, strategies and methods. This isn't a comprehensive list of all the changes made, nor is it a description of the final environment, both of which are beyond the

scope of this paper. Instead, it's a look at the process of change and the lessons we learned.

## Building the New Environment

The changes in the computing environment took place over a period a bit longer than one year. This section presents some of the important events in that process.

### Formed a Team (July, 1992)

The Dean of the College made a decision to hire a UNIX administrator who was familiar with the Internet community. This position was to report directly to the Dean, not to the faculty, and was to manage the Systems Group. This person was given nearly complete autonomy over any matters related to the computing environment in the College. I was hired for this position.

Another position was created for a 'UNIX Systems Programmer', hired about a month later. This person was expected to provide the services expected of an Intermediate Systems Administrator.

These positions were in addition to two staff members, one of whom had been recently hired as a Technician. These four people formed the Systems Group, which consisted of three new hires.

This was a radical and important change for the College. Up until this point, technical positions had not been targeted for people with UNIX expertise, and had not been considered to have positions equivalent to the faculty. It was very clear who had what responsibilities and who reported to whom. By having one boss (the Dean), instead of twenty-five (the faculty), I could concentrate my efforts on fixing the environment rather than running to answer the requests of twenty-five people.

### Examined and Marginally Stabilized the Computers (July, 1992)

At this point, we explored our surroundings and found all sorts of interesting things. No one knew what computers existed. No one knew quite how the network was arranged. No one was sure what software was installed. Hackers had infiltrated the network months earlier but were thought to be gone. We found seemingly endless technical nightmares, such as there being exactly one NIS [1] server for 100 UNIX hosts.

We fixed a few critical problems, such as removing routing loops, which undoubtedly cut down on the number of crashes per week. At that point we casually estimated that it would probably take all summer to get the environment in shape.

We were wrong.

### Punted (August, 1992)

We spent a about a month that summer cleaning up the major problems, and gradually became convinced that we might have bitten off more than we could chew. Too many things were unknown, broken, or both. For example, a set of disk servers were missing critical parts of the OS (like /bin/my, and half of the /usr tree). There were no copies of the OS on tape or CD, so what was installed was all we had to use. And, of course, the disks on those servers were going bad. The servers (all of them) rebooted frequently.

Nor were the servers the only problem... Security was still an open question. Critical network cables were far out of specified parameters. Every day brought new delights.

At this point it became obvious that this wasn't going to be a simple fix and wasn't a standard upgrade. No part of the system could be relied upon. The whole network, from the cables all the way to the software, drastically needed to be rebuilt. In most situations, one must work within the existing structure to improve it. We didn't feel that would be possible here, since we couldn't trust any part of it... two years down the road, we might still be wondering what important part of /usr was missing.

This was a unique opportunity. We would build a completely new computing environment that had nothing whatsoever to do with the existing one. We would essentially be building a new computing environment from scratch, except that we would be doing it for a hundred computers and an existing user base. Unimaginatively, we named the project "Newnet".

### Newnet Was Born (September, 1992)

We used funds from a research grant to buy a SPARC ELC. We also discontinued hardware support for computers that no longer needed support (some of which no longer even existed) to buy two additional SPARCs. Each of these came with their

own copy of SunOS 4.1.2 on CD-ROM. It was quite refreshing to have a copy of an operating system that could be used to rebuild a computer if a disk crashed.

We installed these computers from scratch and connected them together. They shared nothing more than a piece of thin ethernet with the existing environment, which, for obvious reasons, had come to be called "Oldnet". (Fortunately for us, ethernet snooping hadn't become popular at the time we did this.)

Our goal at this point was to use these three SPARCs to build a working software structure that everyone could eventually move to. We didn't have a formal plan in place. In hindsight, we should have analyzed our needs more carefully at this time.

### Designed A Consistent Directory Structure (September, 1992)

The first step in building the network was to create the file system for the network. Rather than haphazardly mount file systems from our machines on each other, we designed a mounting and naming scheme that we felt would scale to several hundred computers and several different machine types.

The details of the plan are beyond the scope of this paper, but a few of the principles may be of interest:

- Local disks are mounted under /export if they will be exported.
- Network disks are mounted under a /net hierarchy.
- We created a global directory named /ccs to be used as a platform independent hierarchy. It has turned out to be enormously useful.
- The network file system structure is identical on every computer, which greatly simplifies navigation from a user perspective.

These assume a disk environment based on local disks and NFS [2], but the basic goals and the naming scheme for the user space of the file system would map to other disk implementations as well. The goals supported by this implementation include:

- A documented file system organization for user home directories and project space, which is consistent across all machines.
- A mechanism for locating files which are only useful only to a specific machine, files which are specific to a type of computer architecture, and files which are useful across all types of computing hardware.

As far as possible, the environment and operating system structure that each machine vendor supplied was left untouched. Other abstractions were used to allow users to quickly and easily move between machines from different vendors.

In order to create the global directory structure, we chose to avoid the automount program provided

with SunOS and used the Berkeley 4.4 automounter, "amd" [3], instead. It had several features that we found to be extremely useful (the /home map, for example), was portable to all of our computer types, and was more reliable.

We completely documented the naming scheme for the local and global disk mounts. After two years of use, the scheme has undergone a few minor revisions, but has generally held up.

#### **Designed A Cloning Process (September, 1992)**

At this point, we had made fundamental changes to how the machines looked after a fresh operating system install. We felt that it was vital to record the changes that we made so that we could replicate them as needed.

We reinstalled one SPARC from scratch from the CD-ROM with the OS on it. We examined the changes that we had made to customize it, and wrote a flexible script to make (and verify) those changes for us. After a few iterations, the script worked quite well. This gave us a tool to take a completely clean SPARC and make it fit exactly into our environment with a minimum of effort.

Every time that we make a change on a computer that should happen on every computer on the network, we adapt this script to make that change for us. For example, we like to install a replacement for "inetd" on our SPARCs and DECs. When this script runs, it replaces the old version of inetd with the new. This means that every time we bring up a new computer, it immediately has all the modifications that the other machines on our network have. Installing a new computer usually takes about 15 minutes of person-time.

Over time, the script has grown and changed. It now consists of a hierarchy of flexible scripts, has a method for dealing with customized configurations of individual machines, and can do kernel installs. While it serves its purpose, we feel that we should move to a more robust and intelligent configuration management scheme in the near future.

#### **Fixed Name Service (September, 1992)**

We now turned our attention to becoming a network of computers rather than a bunch of computers connected by a network. The first step in this move was to get name service functioning correctly.

Up until this time, the College, and indeed the whole University, had been doing hostname lookup with a gigantic hosts file. The problems with this scheme are well-documented [4]. We can verify that those problems are real. In addition, reverse name lookup failed for all of the computers in the College.

We designated one of the SPARCs to be the nameserver for the College, and installed the latest version of the Berkeley Internet Name Domain software. We worked with the network authorities for the University, who, fortunately for us, were

happy to delegate name service for our domain name(s) and our network addresses to us. Name servers are handy tools, and having control of your own name server is a good thing.

Once name service was working, we modified the Newnet hosts to do hostname lookups correctly.

This fixed the naming problems for Newnet, but Oldnet was still broken. While we wanted to limit the amount of work we put into Oldnet, we also felt that having name service working correctly could only make it easier to administrate Newnet. We therefore modified Oldnet to use the nameserver on Newnet, and quit using host files on Oldnet as well.

This was the first instance of what was to become a regular and important tactic as Newnet grew: we used the work we were doing on Newnet to help support Oldnet.

#### **Fixed Mail Service (September, 1992)**

At this time, the College's mail setup was not homogeneous. Mail to different machines ended up in different locations, and outgoing addresses had different appearances. Mail bounced a lot.

We followed the advice of the O'Reilly books [5] and created one mail hub for the College. At the time, we put it on the same machine as the nameserver.

We made outgoing mail appear to come from the domain, and put in MX records that were used to direct all mail to Newnet to the mailhub.

Continuing the tradition of supporting Oldnet with Newnet, we eventually setup MX records for Oldnet hosts pointing to Newnet and forwarded mail to the appropriate Oldnet computers from Newnet. Oldnet computers were configured to send mail to Newnet's mail hub for delivery.

This was the most important thing we did that year for the users of Oldnet. Email, which is probably the most important function of the computers in the College, became reliable and much simpler. This showed the Oldnet users that progress was being made towards the eventual goal.

#### **Defined a Server Strategy (October, 1992)**

We now had a name server, a mail server (the same computer), several disk servers, and perhaps one client machine. It seemed like a good time to consider computer roles and uses.

We decided to designate a computer as a "server" if it provided critical network services. We made a rule that only members of the Systems Group were allowed to login to servers, and that those machines should not be used for general processing. This policy was approved by the faculty resources committee.

This policy has become something we rely on — when a server crashed, we could be sure it wasn't

from user code. When a user's process started spawning across computers, it didn't affect the servers.

#### **Designed a Coherent Software Installation Method (October, 1992)**

Now it was time to start building the software base. We had been installing software by following unwritten rules. Knowing the role of servers and the directory structure, we felt that it was time to work out a consistent and reasonable software installation method. It turns out that we hadn't quite been following the unwritten rules, so formalizing the process was important.

The important aspects of a software installation plan included software naming, installation history, version control, documentation location, and architecture dependent and independent directories. Similar structures can be found in the LISA archives. LUDE [6] and Depot [7] are good examples of software structures. We chose not to follow these plans because they didn't fit into our plans or didn't fit our needs, but our solution is similar to those.

Over the next several months, we built a set of tools that eased software installation significantly. It was possible to build these tools because we had documented our directory organization and our software installation methods.

#### **Enlarged the Team (October, 1992)**

A number of students in the school had become very interested in what was going on. We formed a small group of volunteers who would help install software and design Newnet. All of them were relative UNIX novices, but had a lot of energy. Newnet development became something that sometimes went all night and carried on into the weekends. We picked a few machines from Oldnet, rebuilt them from scratch, and continued to install the basic software base. The students didn't remain UNIX novices for long.

Each volunteer was given root access to all of Newnet when they (and I) felt comfortable with the concept. At the time, Newnet was pretty small and the group worked closely, so the security risk was minimal. Change management was more of a problem, but one of the first things one of the volunteers did was install RCS [8], a version control system.

As a side note, we still have student volunteers, and several of them have root access, but we are a bit more restrictive with who may have root. For example, we limit some tasks using sudo, a publicly available program that allows one to specify who may execute what as root.

#### **Oldnet Crashed (October, 1992)**

Oldnet, which was the primary computing base for nearly a thousand users, was still struggling along pretty much as it had been in the summer. A hardware problem on the only NIS server caused

downtime for the whole network for two days. A few hours into working on it, we realized it was serious problem and developed a new plan for Oldnet.

One of the client computers on Newnet was turned into a server and used as the NIS server for Oldnet. Other Oldnet computers were coerced into being NIS servers for each of their respective subnets.

We took the downtime opportunity to trace cables under the machine room floor. We made the first version of a network map that we had seen for the College, and, while we were at it, removed over 150 cubic feet (in piles) of unused cable from under the floor.

#### **Defined Newnet Clearly (November, 1992)**

Having come this far, we understood the concepts behind Newnet better. We had been building an independent network with no real thought as to how to handle the conversion from Oldnet to Newnet, or how to manage the two networks simultaneously. We weren't exactly sure what a Newnet computer was, nor were we clear on who could use it.

We developed a document that defined a Newnet computer. It had to have been built from scratch from a clean copy of the operating system. It had to have been modified with our cloning script which included kernel modifications, security fixes, and OS patches, among other things. It had to be a member of the Newnet NIS domain, and was therefore limited to the (small) set of Newnet users. It would only use Newnet servers. For security reasons, Newnet users were not allowed to login to Newnet from Oldnet. Therefore, one could not have a login on Newnet without first having their workstation rebuilt to be on Newnet.

Having a clear, written definition served an important function. The process of writing it helped us understand what was important about Newnet. Once we had a functional definition, we could see what we had to do next, and could make estimates of how long it would take.

#### **Addressed the User Issue (December, 1992)**

Once we understood exactly what Newnet's goal was, we tried to project a timeline. It became obvious that it would be several more months. We had a few problems with this. First of all, the users on Oldnet were getting tired of being there and putting up with the problems. Further, it wasn't technically obvious how to move non-technical users from Oldnet to Newnet.

We made two important decisions:

1. We would update the user community relatively often as to the status of Newnet. This would let them know that progress was being made.

2. We would share home directories between Oldnet and Newnet because it was technically very difficult not to. We didn't have enough disk space for everyone to have two homes, which confuse most of the users, anyway. Unfortunately, our home disk servers were on Oldnet, so this went against our policy that Newnet computers shouldn't use Oldnet servers. However, we had no choice.

In order to keep Newnet reasonably secure, we continued to enforce the policy not to allow logins to a Newnet machine from Oldnet. The reverse was not true at all, however. In fact, the members of the Systems Group liked Newnet too much to use Oldnet, so we were almost completely administering Oldnet from Newnet.

#### **Enlarged the Population (January, 1992 - March, 1993)**

Newnet continued to grow, both in terms of the number of users on it as well as the number of computers on it. The users were limited to the Systems Group, the student volunteers, and certain key people who needed Newnet accounts for technical reasons or for morale. For example, the Dean of the College was moved to Newnet, as was the Assistant Dean. We watched these moves closely, noting the problems that they had, and working out scripts to help people move from Oldnet to Newnet. After two non-technical users had migrated, we wrote a much-needed (but minimalistic) user's introduction that described differences to expect. ("Emacs works. Your shell works. X works. Suntools doesn't. You can find software. ...")

#### **Designed the User Environment (March, 1993)**

On Oldnet, the primary means of setting up one's environment was to copy someone else's dotfiles and hope they worked. Most people used the ones developed by a knowledgeable professor. This caused all sorts of problems. (Without the professor's dotfiles, they would have been much worse...)

On Newnet, we built a set of reasonable default files that were installed in new accounts. We spent a lot of time on this. They were fully documented and contained lots of examples for users who wished to modify their environments. While the dotfile situation on Oldnet caused enormous trouble, we've had very few problems on Newnet with user environment configuration.

In addition, we designed an environment abstraction mechanism that allowed the users to select what sets of software they would like to use. The user's PATH, MANPATH, and related environment variables were built based on the user's selections. The mechanism has allowed maximum flexibility for users and administrators. We were able to do this because we had a consistent software installation scheme. This software mechanism is

described elsewhere in these proceedings. The most important aspect of this approach is that it built an abstraction between the user's environment and the software installation environment. They wanted to modify one, we wanted to modify the other.

#### **Changed the Domain Name (April, 1993)**

Up until this point, the network domain name for Northeastern University had been "northeastern.edu". After months of politicking on our parts, it was changed to "neu.edu", which is considerably easier to type.

While the name change wasn't a vital part of Newnet, it certainly fit in. We were changing everything else about the network, so why not change the domain name?

The Newnet computers were moved to the "neu.edu" domain overnight, because administering them was easy. Oldnet computers were updated as they moved to Newnet. It became obvious from the name of the computer which net it was on. Users and administrators liked that.

#### **Implemented the Hosts Database (April, 1993)**

The domain name change gave us the opportunity to finish a set of tools based around a hosts database. They were used to build any network or host configuration files associated with IP addresses, including nameserver files, bootparams, ethers, printcap, hosts.lpd, hosts.equiv, and xdm configuration files, to name a few. The scripts did sanity checking on all the files before installing them to make sure that the data was reasonable. They were also used to build the list of computers that various user groups can access.

Once again, we used these same scripts to maintain parts of Oldnet.

Host configuration files related to IP addresses had been an enormous problem on Oldnet. Changing IP addresses or hostnames of a computer had been done by hand. The frequency of operator error in the midst of Oldnet's chaos was pretty high, causing all sorts of interesting problems. By using these host file configuration scripts, we virtually eliminated the chance for human errors. We've had absolutely none of these types of problems since.

Our experience here has defined a technique that we try to follow as much as possible - we automate anything that we do more than once, and we do sanity checking on files and systems before we install them.

#### **Continued to Support Oldnet (May - June, 1993)**

By this point, Newnet was providing all of Oldnet's critical network services. We had three major steps to complete before replacing Oldnet:

- We needed to complete the Ultrix environment.
- We needed to write reasonable documentation for users about how to navigate Newnet.

- We needed to figure out how to migrate 1000 user accounts to Newnet and to write the tools to help us to that.

In order to keep users pacified while we continued to develop Oldnet, we exported the Newnet software environment to Oldnet, and explained to the user community how to modify their PATHs to get to it. We also moved several more faculty members and their graduate students to Newnet to use as beta-testers.

#### **Wrote User Documentation (Summer, 1993)**

Based on responses and questions from the users that had moved to Newnet, we enhanced our user documentation.

The first thing that we wrote was a Newnet users guide that explained the few differences that users would actually have to handle. This was really a more detailed version of the first handouts we had prepared. We turned a version of this into a Frequently Asked Questions file and posted it to various local newsgroups.

We developed a WWW-based help system that documented all of the software installed, organized by category. Again, this was possible because we had a well-defined software installation method. This was intended to help users locate software that might be useful to them, and to partially organize software documentation.

A comprehensive user's guide would have been useful, but we felt that most of the students on the network wouldn't read a 20-page document that explained how to print or how to access the local modems. Instead, borrowing an idea from the University of Oregon, we wrote several one-page documents, each of which were about specific topics. These papers, some of which talked about about dotfile customization, lab usage, simple unix commands, and emacs, were promptly named Clue Sheets. We distributed these in the public computing labs.

#### **Developed The Account System (Summer, 1993)**

In order to manage the creation of a thousand new accounts on our network, we designed a comprehensive account strategy. Each account had to be exactly one of these types: faculty, staff, grads, majors, students, or guests. Each of those had certain requirements that had to be met, including, for example, verification by the dean or with the registrar. In addition each user had to sign a form stating that they would abide by the stated reasonable use policy when using their account. These policies were approved by the faculty resources committee.

We wrote an "account" program that users had to run to request an account. (This could be done by users with no account by logging in as "account".) This program would perform certain accounting functions, such as making sure their requested login

name was permissible and checking their password against a dictionary, and then would put their account request in the creation queue.

On the administrator side, we developed scripts that automated functions such as creating accounts, expiring accounts, changing passwords, and most importantly, moving accounts from Oldnet to Newnet. When an account was moved to Newnet, we added it to the correct NIS files on Newnet, enabled its shell, and put a trigger in the account that would cause new dotfiles to be installed in the account when the user first logged on to Newnet. We were reluctant to directly modify the user's accounts this way, but Oldnet dotfiles simply wouldn't function on Newnet. We compensated for this by notifying the user that it was happening, and storing their original files as backup versions.

Perhaps the most important aspect of this is that we have a documented account system, and we have an account policy that every user has signed. Converting them over to Newnet was a convenient way for us to reach all of the users on our net and have them agree to the policy.

#### **Demise of Oldnet Predicted (September, 1993)**

It was time to open up Newnet to the masses. We had a strategy for moving accounts, we had documentation for them, and we had a functional environment. We announced the upcoming move on local newsgroups and put pointers to those announcements in the message of the day.

Faculty members, their graduate students, and related workstations were the first large group to be moved. We converted about a hundred accounts and a hundred computers in the week between Summer quarter and Fall quarter.

Simultaneously, we converted all of the public laboratory computers and about half of the central computing facilities to Newnet in preparation for the next quarter. We left the other half of the central computing facilities on Oldnet for people who might not get around to moving to Newnet right away.

Once Fall quarter started, we allowed all of the remaining Oldnet users to register for Newnet accounts (assuming they met the requirements). Within a week, we created another six hundred accounts on Newnet.

We were expecting an onslaught of problems and complaints, but for the most part it went very smoothly. The primary difference between a Newnet account and an Oldnet account (beyond the fact that it worked well) was the computer that the user logged into. Their home directory was the same, and their dotfiles had been massaged by the account program, so their environment on Newnet was quite satisfactory. We believe that things went as calmly as they did because we tried to make as few obvious user-level changes as possible, and that

we had a lot of documentation available explaining those changes.

These were the key points to our final conversion strategy:

- We were going to disturb a lot of users, so we made a lot of noise about it. (No matter how much noise you make, most of them will be surprised, but at least you can point at your signs.)
- Once we committed to it, we did it as fast as we could in order to minimize confusion.
- We left a way out. There were a few machines on Oldnet, just in case.

#### Settled Down (Fall, 1993)

Once the majority of users were on Newnet, we were able to stop supporting Oldnet (almost). This gave us time to finish off a few more projects which enhanced Newnet.

We installed new printers and a printer quota system. Both of these had been desperately needed for quite some time.

We rebuilt the physical network from scratch, using 10Base-T ethernet instead of the thick and thin net combination. We moved to SNMP manageable 10Base-T hub units, real routers, and a reasonable network configuration. This drastically improved performance and reliability. We could have made the shift to new wiring while still operating Oldnet, but it would have been nearly impossible to reconfigure the machines and the network addressing. On Newnet, it was relatively simple and resulted in about one day of downtime while we switched all the servers and reorganized the backbone topology. Once the servers were happy, we went back to our tried-and-true technique of incrementally moving machines from one wiring structure to the other while supporting the old wiring with the new.

And of course, we installed a lot of software needed by users that we had neglected. The users will always find things that you haven't done.

#### Matured (Spring, 1994)

The last Oldnet user machine was converted to Newnet in December. The last Oldnet server (which was being used for Usenet news) was turned off in March. Except for the network rewiring, the majority of Newnet has been up ever since it was conceived.

Newnet continues to be developed. Current projects include:

- Documentation for the systems administrators.
- A request tracking system.
- Proactive network management.
- A server for machine configuration files (eventually targeted as a replacement for "ccsify").
- Figuring out what to do next now that the big goal has been achieved.

Since it's no longer the "New" net, we refer to these developments collectively as "Tenwen".

#### Fundamental Concepts

While building Newnet, we followed several principles that we felt were essential to our success.

- Consistency and well-documented procedures are critical.

By being consistent and documenting our design of important systems, we were able to create programs that automated important functions, and we minimized mistakes.

One example of this is our directory structure and disk mount point scheme, which we relied upon when creating disk management programs.

In addition, our account types and policies and our software installation methods follow the rules we established for them. We have been able to write programs that automate software installation and account creation which also rely on their being a consistent directory naming scheme.

- Automate anything you do more than once. Among other things, we've automated machine configuration, account creation, and the generation of files related to IP addresses. An essential part of all these automation tools are sanity checkers that minimize the opportunity for mistakes.
- When building a new system that will replace an existing system:
  - Create the new system separate from the old system.
  - If they will both exist for a while, support the old one with the new one.
  - Announce the demise of the old one well in advance of really discontinuing it.

For example,

- When Oldnet's NIS configuration died, we supported it with Newnet.
- We supported the old wiring with the new 10BaseT wiring.
- The Newnet software environment was exported to Oldnet.
- The old domain name was gradually converted to the new domain name.
- Newnet itself was built this way.

We've found that this makes the old system more reliable while testing the new system. In addition, it eases the transition, and gives you the chance to move back to the old system when necessary.

- Have formal, written policies.

We have policies about the uses of our servers, uses of disk and printer resources, who may have root access, what kinds of computers we support, and who may have

accounts on our network. The faculty resources committee has agreed with and supports these policies, and they are available for reference. In the case of the account and resource policies, users have to sign that they have read and will follow these policies.

- Build in abstractions that you control.

Abstraction is one of the basic tools of computing. By creating a well-known interface and supporting that interface, one is given the freedom to change the underlying implementation. This is the core philosophy of everything from Turing machines to objects, and it applies to systems administration as well. There are several such examples in Newnet. Domain-based mailing, where one sends mail to a domain rather than to a host, was one of the first such abstractions put in place in Newnet. The advantages of doing this are widely documented.

Our automount scheme mounts directories into a /net hierarchy, and then most user file systems are built with symbolic links into /net. We can change the /net mounts at any time as long as we update the interface, i.e., the symbolic links.

The software environment mechanism that we use allows us to freely change the location and version of software packages. The user interface is a list of categories of software that they wish to have in their environment. That is expanded into a set of environment variables whose details we control.

- Support and communicate with the user community.

It would not have been possible to put the effort into Newnet had the users not been willing to wait for it.

As we built Newnet (and mostly ignored Oldnet), we periodically announced the progress of Newnet. We made the Newnet software available on Oldnet when it became obvious that Oldnet would be around longer than we hoped. Important and vocal users were moved to Newnet early on in order to test Newnet, keep them satisfied, and get them on our side.

### User Reaction

User reactions during this whole process were mixed. While we were building Newnet, most users were curious about it. Most of the questions we had were along the lines of "What are you doing, again?", and "How's it going?". Towards the end there was a definite sense of impatience. Overall, the faculty and the user community were amazingly supportive of the whole process.

We expected a lot of negative reactions from users once we made the switch. After all, we were

changing their whole environment, and they might not have agreed with our opinions about Oldnet. However, we only got two kinds of negative responses. The first were from the set of users who didn't qualify for accounts on Newnet, either because they were no longer students or had never been. They were understandably upset because they were losing their accounts. The second complaint type was related to software that wasn't installed on Newnet. We tried to fix these as soon as we could, although in some cases we slipped and installed things much later than we should have.

The only other complaint at all came from a user who didn't like the naming scheme we had chosen for Newnet. On Oldnet, most of the computers were named "sun3140a", "sun3140z", "dec5200a", and so on. We wanted something a bit more lively for Newnet, so we named all the Newnet SPARCs after mountains, and the DEC's after features of the Northeast, to name a couple of our themes. This user complained that he could no longer figure out what computer to login to based on its name. We told him about the "computers" command, which listed all the available computers, their machine types, their load, and their location. He was satisfied.

Other than these complaints, the user community was very enthusiastic. We got a lot of positive email (which, in our profession, one learns not to expect), and one particularly wonderful faculty member dropped off a box of six dozen chocolate chip cookies for the Newnet crew.

### Regrets

The transition to Newnet has been a major success, but we could have improved the process in several ways.

Without question, we should have written a lot more documentation while we were designing. We wrote a lot, but not nearly enough. Now we need to completely document the account system, the NIS methods, the mail system, and the security policies, simply to name a few.

Building Newnet took a lot longer than we expected, partially because we didn't know exactly what to expect. A clearer definition of the goals right from the start would have helped us focus and plan. On the other hand, we felt at the beginning that we could have installed Newnet in less than a month, and perhaps we could have, but it would not have been the solid and cohesive system that it is now.

Finally, supporting two environments for as long as we did was actually quite difficult for the administrators who weren't intimately involved in the design of Newnet. There were always two different ways to do things, and the potential for confusion was high. We should have had better

documentation for procedures, or perhaps spent a bit more time trying to make Oldnet easier to administer.

### Conclusion

In a bit more than a year, we moved from a dysfunctional environment to a very productive one. We did so by building the new environment separately from the old. The new environment, through the use of consistent naming, abstraction mechanisms, and documented policies, was designed to be easy to administer and use. While we developed the new environment we were able to use it to support the old one. We recommend this approach to anyone that has the option. Even where it may not be possible to follow our suggestions, we feel that the lessons we have learned will be valuable.

### Author Information

Rémy Evard has been the leader of the Experimental Systems Group at Northeastern University for two busy years. He received his M.S. in Computer Science from the University of Oregon in 1992, where he developed many of the basic concepts for Newnet while working as a graduate student systems administrator. While leaving behind the bicycles and trees of Oregon was a traumatic experience, he feels that the in-line skating in Boston almost makes up for it. He may be reached electronically at [remy@ccs.neu.edu](mailto:remy@ccs.neu.edu).

### Acknowledgements

Newnet was a long and exhausting journey that took place mostly after midnight and on weekends. It couldn't have happened without a lot of help.

Students who put everything they had into the design and implementation of Newnet include Brian Dowling, Ivan Judson, Robert Leslie, and Matthew Wojcik.

People from the University of Oregon and Argonne National Laboratory who had who helped in the initial design include Paul Bloch, Bart Massey, Bill Nickless, and Robert Olson.

The staff who coped with Oldnet while simultaneously contributing to Newnet consisted of Tom Coveney, Lorraine Gabrielle, and Jim Mokwa.

Finally, a major thanks goes to the faculty of the College of Computer Science and to Michele Evard, for hanging on while Newnet was developed.

### Bibliography

- [1] Sun Microsystems, "The Network Information Service," in *System and Network Administration*, pp. 469-511, Sun Microsystems, 1990.
- [2] Sun Microsystems, "Network File System: Version 2 protocol specification," in *Network*

*Programming Guide*, pp. 168-186, Sun Microsystems, 1990.

- [3] Jan-Simon Pendry, "AMD - An Automounter", Department of Computing, Imperial College, London, May, 1990.
- [4] Paul Albits and Cricket Liu, "DNS and BIND", O'Reilly & Associates, Inc, 1992.
- [5] Hal Stern, "Managing NFS and NIS", O'Reilly & Associates, Inc, 1991.
- [6] Michel Dagenais et. al, "LUDE: A Distributed Software Library", in *LISA VII Proceedings*, pp. 25-32, Monterey, CA, 1993.
- [7] Walter C. Wong, "Local Disk Depot - Customizing the Software Environment", in *LISA VII Proceedings*, pp. 51-55, Monterey, CA, 1993.
- [8] Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System", in *Proceedings of the 6th International Conference on Software Engineering*, pp. 58-67, ACM, IEEE, IPS, NBS, September, 1982.

### Appendix A: Literature

We found many sources of information to be invaluable during the design and creation of Newnet. The articles, books and journals that we read and borrowed ideas from are too numerous to list. However, these were the main sources of inspiration and information:

- The SAGE News area of ;login:, the USENIX Association Newsletter, contains page upon page of good advice.
- The LISA Workshop and Conference Proceedings.
- The complete UNIX System Administrator series of manuals from O'Reilly & Associates.

**Appendix B: Time Line**

This has been included because some readers may find it useful to refer to while perusing the paper. It's also interesting to note what order various changes took place. In particular, one can see that we built a foundation of core network services, enhanced the environment, and then adapted it to the user population.

- July, 1992 - Assessed the situation. Miscalculated.
  - August, 1992 - Decided to build an independent network.
  - September, 1992 - Started three SPARC with clean OSs.
    - Designed the global file system.
    - Wrote cloning scripts.
    - Built a name server.
    - Built a mailhub based mail system.
    - Built the NIS environment.
  - October, 1992 - Defined a policy for servers.
    - Defined a software installation strategy.
    - Recruited students.
    - Started supporting Oldnet with Newnet's NIS servers.
  - November, 1992 - Defined Newnet clearly.
    - Installed a lot of software.
    - Designed the WWW-based help system.
  - January, 1993 - Added the first non-systems user to Newnet.
    - Converted a few more machines to Newnet.
  - February, 1993 -
  - March, 1993 - Designed the basic user environment.
  - April, 1993 - Changed the domain name.
    - Created the hosts database and IP config file manager.
  - May, 1993 - Built the first Newnet DEC.
  - June, 1993 - Built the first Newnet Sun3.
  - July, 1993 - Wrote user documentation.
  - August, 1993 - Wrote the account system.
  - September, 1993 - Opened Newnet to everyone.
  - October, 1993 - Closed down Oldnet to users.
    - Installed software that we'd forgotten about.
  - November, 1993 - Stopped to breathe.
  - December, 1993 - Moved to 10baseT wiring and a new network organization.
  - March, 1994 - Shut off the last Oldnet server.
- Throughout this whole period, we installed software. At last count, there were approximately 1500 programs in the default path for SPARC users.

# Kernel Mucking in Top

William LeFebvre – Argonne National Laboratory<sup>1</sup>

## ABSTRACT

For many years, the popular program *top* has aided system administrations in examination of process resource usage on their machines. Yet few are familiar with the techniques involved in obtaining this information. Most of what is displayed by *top* is available only in the dark recesses of kernel memory. Extracting this information requires familiarity not only with how bytes are read from the kernel, but also what data needs to be read. The wide variety of systems and variants of the Unix operating system in today's marketplace makes writing such a program very challenging. This paper explores the tremendous diversity in kernel information across the many platforms and the solutions employed by *top* to achieve and maintain ease of portability in the presence of such divergent systems.

### Motivation

Any system administrator knows the litany. A line of users start forming outside of the office, the phone starts ringing off the hook, and everyone has the same thing to say: "this lousy computer is taking minutes to do anything, even a simple *ls* command." Most experienced administrators will look for the same thing: a cpu-intensive process that is tying up most of the computers cycles. Perhaps they get their information from the standard Unix *ps* command, or perhaps they will start with *uptime* or *w*. Many administrators "in the know" will use the freely available software package *top*. In any case, the system administrator is seeking information that only the kernel has: what is the status of the computer's resources and which processes are using them?

For all but the most modern versions of Unix this information is only attainable by reading it directly out of kernel data structures. Imagine the task of writing a program which extracts this information and designing it to maximize portability. When you consider the incredible variety of hardware platforms and Unix variants in the marketplace, the job seems almost insurmountable. Kernel designers don't often consider ease of access to information by outside processes when designing internal data structures. Consequently, even minor operating system revisions may make changes which have a major impact on kernel-dependent programs. Although not completely successful, the design employed by *top* has achieved a reasonable medium

between the needs of extracting useful information and maintaining ease of portability.

### A Top Process Display for Unix

The software package *top* presents a full-screen display of the top cpu-using processes on the system. It also presents some essential system information about cpu cycles (system versus user), memory usage, load averages, process categories, and other tidbits. This information is updated regularly (usually every 5 seconds). Refer to Figure 1 for a sample display from *top*.

The display varies depending on the particulars of the underlying operating system. The sample shown was taken from a system running Sun's Solaris 2.3. In general, the top four lines show information about the overall health of the process environment. The first line shows the 1, 5, and 15 minute load averages. The second line shows the total number of processes and how they break down in to separate categories (such as sleeping, running, and stopped). The third line shows percentages spent in each cpu state: this is the line that will show when a cpu is spending a disproportionate amount of time in the kernel. The last line shows information about memory usage.

The remainder of the display consists of information about each individual process. Again, this information will vary depending on the operating system, but in general it will show the process id, username of the owner, internal priority, *nice* setting, total virtual memory size, amount of virtual address space currently in physical memory (the "resident set" size), process state, cpu time, cpu usage percentages, and command name. The display is sorted by one of the cpu percentages so that the top percentage using processes are shown first.

This information is updated regularly, usually every five seconds by default. The user can set the update time to any number of seconds (including

<sup>1</sup>This work was not funded through Argonne National Laboratory. The submitted manuscript has been authored by a contractor of the U.S. Government under contract No. W-31-109-ENG-38. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

zero, in which case updates happen continuously). Other options are also available to regulate a variety of items.

Usually there is only one way for *top* to obtain all the information it displays. It has to dig around in the kernel for it. This is the same way that Unix commands like *ps*, *netstat*, *vmstat*, and other status-displaying utilities obtain their information. The author has affectionately coined the phrase *kernel mucking* for this procedure. Despite any sugar coating that may be available in the libraries, in the final analysis there is usually only one way to get data out of the kernel: by using *open*, *lseek*, and *read* on the device */dev/kmem*. This device is a very special character device. The kernel maps accesses to this device so that they exactly correspond with kernel memory itself. Any byte in the kernel can be read from somewhere in */dev/kmem* (and if the device is opened for writing, any byte can be written as well).

Some Unix System V release 4 systems provide an easier way to get per-process information: the pseudo file system */proc*. In fact, the current trend in Unix releases is to eliminate the need for mucking around in the kernel as much as possible. The *proc* file system is one step in that direction. Some SVR4 versions now also have other hooks by which programs can get non-process related information about the system, such as Sun's *kstat* device.

BSD Unix version 4.4 has the *sysctl* system call which provides a cleaner interface to kernel data. One of a number of structures can be requested. The kernel will fill in the appropriate data and return it to the requesting program. This is

obviously much easier than kernel mucking, but is limited to only the information that the kernel writers saw fit to provide. If you want to access a value that is not provided by one of these structures, you have to resort to old fashioned means.

For most Unix variants, there's still only one way to get to all this information: kernel mucking. Obviously, any program which needs to do this is going to be very dependent on the specific organization of the kernel running on the machine. That means the task of writing such a program to be portable across different versions of Unix (and thus different architectures) is going to be extremely difficult. In fact, just making it portable across different minor revisions of the *same* version of Unix is difficult. One of the primary design goals for version 3 of *top* is to isolate all the machine-dependent code in one source file and to provide a clean and well defined interface as a set of functions. All the functions which handle options processing, screen management, display updates, internal commands, etc., are all part of the machine-independent portion of *top*. These functions make calls as needed on routines in the machine-dependent file library. Only the latter need concern itself with kernel mucking and other specifics of the operating system.

Version 2.7 of *top* (before all the machine dependent code was isolated) was difficult and time consuming to port: it only ran on a handful of different systems. The reorganization of version 3 made *top* significantly easier to port. As a consequence, ports now exist to allow *top* to run on these platforms:

```
last pid: 2980; load averages: 0.08, 0.14, 0.11      11:12:21
58 processes: 56 sleeping, 1 stopped, 1 on cpu
Cpu states: 93.1% idle, 3.1% user, 3.9% kernel, 0.0% iowait, 0.0% swap
Memory: 21M real, 1424K free, 43M swap, 52M free swap
```

PID	USERNAME	PRI	NICE	SIZE	RES	STATE	TIME	WCPU	CPU	COMMAND
2980	root	7	0	1692K	1412K	cpu	0:01	0.77%	4.25%	top
709	lefebvre	28	0	9900K	2612K	sleep	21:13	0.13%	1.93%	Xsun
741	lefebvre	28	0	3404K	1700K	sleep	0:36	0.08%	0.77%	cmdtool
787	lefebvre	14	0	11M	2180K	sleep	6:11	0.00%	0.00%	maker4X.exe
93	root	24	0	1772K	760K	sleep	1:27	0.00%	0.00%	automountd
1	root	34	0	696K	96K	sleep	1:23	0.00%	0.00%	init
70	root	34	0	2184K	600K	sleep	0:39	0.03%	0.00%	keyserv
68	root	29	0	1608K	588K	sleep	0:35	0.03%	0.00%	rpcbind
892	lefebvre	-25	0	2376K	968K	sleep	0:27	0.00%	0.00%	xmh
1213	lefebvre	14	0	5024K	324K	sleep	0:14	0.00%	0.00%	emacs
716	lefebvre	24	0	1844K	684K	sleep	0:11	0.00%	0.00%	olwm
2967	lefebvre	24	0	4960K	3836K	sleep	0:07	0.00%	0.00%	emacs
2703	lefebvre	34	0	3388K	1332K	sleep	0:04	0.00%	0.00%	cmdtool
86	root	-25	0	1560K	488K	sleep	0:04	0.00%	0.00%	inetd
1255	lefebvre	34	0	3404K	1100K	sleep	0:03	0.00%	0.00%	cmdtool

Figure 1: Sample output from *top* running on Solaris 2.3

386BSD	Intel-based SVR4.2
AViiON w/DG/UX 5.4+	Mt. Xinu MORE/bsd (VAX)
BSD/386	NetBSD
Dynix 3.0.x	OS/MP 4.1A (Solbourne)
Dynix 3.2.x	SunOS 4.x
generic 4.3BSD	SunOS 5.x (Solaris 2.x)
generic 4.4BSD	Ultron 4.2 or later
HPUX (most versions)	UMAX 4.3 (Encore)
Intel-based SVR4	UTek 4.1 (Tektronix)

Ports are in the works for the DEC Alpha machine and for IBM's AIX.

When describing detailed aspects of kernel mucking, this paper will try to remain as generic as possible, but this requires dealing in vague generalities. In some cases, specific examples are used to make a particular point. There is no guarantee that the example will work on any particular platform or Unix variant.

### Accessing Kernel Information

Many programmers have never been exposed to the specific techniques involved in retrieving information from the kernel. Indeed, many programs have no such need. This section offers a brief overview on the technique.

#### Reading from `/dev/kmem`

It may seem strange to you that you access memory as if it were a disk, but that is essentially what is done. Think of kernel address space as if it were one large file, with the zeroth byte of the kernel corresponding to the beginning of the file. If you want to read the byte at address *x*, you would use a code fragment like this one:

```
int i;
unsigned char c;
i = open("/dev/kmem");
lseek(i, x, 0);
read(i, &c, sizeof(unsigned char));
```

Reading an entire longword is done similarly: the only difference being the use of an unsigned long and the argument to *sizeof*.

#### Finding Kernel Addresses

So where do the addresses come from? The kernel is stored on disk as an executable in the root directory. Depending on the particular Unix variant in use, it could be named `/unix`, `/vmunix`, `/stand/unix`, `/kernel/unix`, or something similar. This is the very same image that is loaded at bootstrap time. It is always stored in the same format as a regular executable, complete with symbol table. This table contains every global variable name along with its address. For an ordinary executable or object file (the format is the same) this information is used by the linker, *ld*, to match up uses of external variables to their definitions. Many installed executables have had this information removed or stripped (usually by the command *strip*), but the kernel image is

intentionally not stripped. The C library contains a function that knows how to obtain the addressing information given a list of variable names: *nlist*.

The C library function *nlist* takes an array of *struct nlist*. Each structure element has room for a variable name (called a *symbol name*), its "value", type, and other tidbits of information. The *nlist* function expects this array to have the name fields already filled in with the names of the variables you want. It then opens the executable of your choice, finds all the values, and fills in the rest of the structure. This value is not the variable's value, but rather the addresses in the executable where the variable is actually stored. Don't confuse these two! The documentation (and the include file) for *nlist* refer to a "symbol" and a "symbol's value." The symbol corresponds directly to the variable's name. The "symbol value," however, is the address in memory where the variable's value is found.

So let's say that we want to find the process id number of the last process that was created. Now, not all Unix variants keep track of this information, but those that do usually store it in a kernel variable called *mpid*. On most Unix systems the compiler prepends an underbar to every external variable name before adding it to the symbol table, so we really want to ask for the variable *\_mpid*. Here are the steps we would take:

- open `/dev/kmem`
- use *nlist* to obtain address for *\_mpid*
- *lseek* to that address
- *read* the value at that location
- display the result

This is fleshed out in Figure 2. For clarity, this example excludes error checking code. In addition to checking the value returned by *open*, the value returned by *lseek* should be checked to insure that it is 0, and the value returned by *read* should be checked to see if it is equal to the number of bytes requested in the read. If either of these kernel calls does not return what is expected, then it is usually an indication that the kernel address used in the *lseek* call is not valid.

```
#include <nlist.h>
int i, mpid;
struct nlist nlst[] =
    {"_mpid"}, {0};

i = open("/dev/kmem", 0);
nlist("/vmunix", nlst);
lseek(i, nlst[0].n_value, 0);
read(i, &mpid, sizeof(mpid));
```

Figure 2: Reading a kernel variable's value

#### The *kvm* Library

Many modern-day versions of Unix make this task less onerous by providing a kernel-access library called *kvm*. Users of this library initially call

*kvm\_open* to initiate use of a given kernel image. This function, rather than returning a file descriptor, will return a pointer to a *struct kvm*, much like the streams library returns a pointer to a *FILE* structure. All other functions in the *kvm* library take a *struct kvm* pointer as an argument and will perform their machinations on the corresponding kernel image. Most *kvm* libraries provide functions to carry out the following operations: open, close, read, write, symbol list (*nlist*), walk through the process structures, obtain process information by process id, obtain a process's user structure. Those who are doing kernel mucking are well advised to use the *kvm* library on any system where it is available. It hides some of the really grubby details.

## Process Structure

Although kernel internals vary widely between different Unix variants, there are some basic concepts shared by all (or nearly all). In the earliest versions of Unix there were two kernel structures employed to keep track of all the information about a process: the process structure and the user structure [4]. This design has been carried through to BSD Unix [3] and System V Unix [2] and is present in every Unix variant seen by this author.

The process structure is also known as the *proc* structure (the name given to the structure is "proc," as in *struct proc*). This structure is typically defined in the include file *<sys/proc.h>*. The information contained in this structure is what the kernel needs to have readily available in order to keep track

---

```
#include <nlist.h>
#include <sys/proc.h>

main()
{
    int i, fd, bytes, nproc;
    unsigned long proc;
    struct proc *pbase, *pp;
    static struct nlist nlst[] = {
        { "_proc" },
#define X_PROC 0
        { "_nproc" },
#define X_NPROC 1
        { 0 }
    };

    /* open kmem, call nlist, get variables' values */
    fd = open("/dev/kmem", 0);
    nlist("/vmunix", nlst);
    lseek(fd, nlst[X_PROC].n_value, 0);
    read(fd, &proc, sizeof(proc));
    lseek(fd, nlst[X_NPROC].n_value, 0);
    read(fd, &nproc, sizeof(nproc));

    /* allocate space for proc structure array */
    bytes = nproc * sizeof(struct proc);
    pbase = (struct proc *)malloc(bytes);

    /* read all the proc structures in one fell swoop */
    lseek(fd, proc, 0);
    read(fd, (caddr_t)pbase, bytes);

    /* iterate thru the result */
    for (pp = pbase, i = 0; i < nproc; pp++, i++) {
        /* display information for interesting processes */
        if (pp->p_stat != 0) {
            printf("pid %d, uid %d\n", pp->p_pid, pp->p_uid);
        }
    }
}
```

Figure 3: Extracting and examining the entire *proc* array

of the process throughout its lifetime. When the process exits and when the process's parent has picked up the exit information (for example, via *wait*), the process structure is freed. Examples of information typically stored in the *proc* structure [3, page 73] are:

- process id, parent process id, pointers to child process structures
- real user id, effective user id
- scheduling: priority (including *nice*), recent cpu utilization, sleep time
- memory management: pointers to page tables and shared program text
- process size (text, data, and bss)
- some signal information

Sound familiar? It should. Just about every piece of process information displayed by *ps* or *top* is stored in the process structure.

The process structures are usually stored in a large array that is allocated at boot time. The size of this array will dictate the maximum number of processes that can be running on the system at any one time. The array elements are also sometimes referred to as process slots. When a process exits, its slot is marked as available. When a process is created, the kernel will hunt down a free slot to use for the new process.

The process array is stored in the variable *proc*. The number of elements (slots) in the array is stored in the variable *nproc*. On a system that has no *kvm* library, you have to use *read* directly to find a specific process or to iterate through the process slots. The *kvm* library will typically include several functions that find and return data in the *proc* array, making such access significantly easier (albeit rather inefficient). These functions would likely be named *kvm\_nextproc*, *kvm\_getproc*, and *kvm\_setproc*.

As an example of reading arrays and structures from the kernel, Figure 3 contains a complete program that reads and iterates through the *proc* array. This program reads the entire array in at once, then steps through it. This is the method that *top* usually uses to read the *proc* array and to read any other large array, as it is far more efficient than performing one read per array element.

#### User Structure

The user structure contains all the stuff that the kernel needs when a process is running (or more specifically, when the process is swapped in). It is defined in the include file `<sys/user.h>`. Unlike the *proc* structure, this structure is not actually stored in fixed kernel memory. Instead it resides in the process's virtual address space. When the process is swapped out to disk, this structure goes with it. The user structure typically contains information [3, page 77] such as:

- execution states (register values and processor status structures)

- open files (file descriptors)
- creation mask (the umask)
- current directory inode
- resources usage information (*struct rusage*) and limits (*struct rlimit*)
- executable "command" name

For most mucking problems, this structure would not be needed, except for the fact that it is the only place where you can get the name of the executable currently running in this process (i.e., the command name). Complicating the matter is the fact that this structure is extremely difficult to find. Since it is stored in the process's address space, it is not readily available to someone who is only mucking around in */dev/kmem*. All that you find there are pointers to the virtual memory page information, which you can then use to track down the physical page addresses, then open another special file, */dev/mem*, and muck around in it for the user structure.

That is, of course, assuming that the process is actually in memory. If it has been swapped out, then an entirely different method must be used to hunt down the swapped out pages in a completely different device: */dev/drum*. The *kvm* library makes this trivial by providing the function *kvm\_getu*. This function takes a pointer to a *proc* structure and does whatever is necessary to retrieve the user structure, passing it back to the caller. Having this function available is especially helpful since the code to obtain the user structure is particularly sensitive to different virtual memory management techniques, and can vary widely between different Unix vendors.

#### Platform Independent Design

In an attempt to isolate as much of the machine dependencies as possible, all of the kernel mucking in *top* is contained in a single collection of functions all residing within one file. Ideally, this is the only file that needs to change when compiling a version of *top* for a different platform. This file, along with some ancillary documentation, forms a machine *module*. At configuration time, a module name is chosen that is appropriate to the platform. All these modules are collected together in one directory. It is interesting to note that these modules comprise about 80% of the total source code for *top*.

#### The Challenge

The crux of the the design is the collection of functions used to obtain the information and their exact definition. The author sincerely wishes that there already existed an OS-independent definition for such a library, but the truth of the matter is that systems vary too much to make such a definition workable. Even if such a definition existed its adoption by just the major vendors would take years. As development of this interface definition progressed, it became clear that the information needs varied too

much between systems and much of the decisions about which statistics to display had to be left to the modules themselves.

An excellent example of this dilemma is the memory status line. In older versions of Unix (BSD 4.2 and SunOS 3), this line displayed: amount of real memory in use, amount of virtual memory allocated, amount of real memory still free. The first two figures were supplemented with the amount of memory used recently (or "active"). These exactly corresponded to fields in the *vmtotal* structure named *total*. But different virtual memory implementations maintain different types of statistics. In fact, SunOS version 4.0 still used the *struct vmtotal* to track some of the virtual memory statistics, but did not fill in the "active" portions of the structure. The original layout of the line appeared as:

```
Memory: 2408K (2560K) real,
        6700K (3202K) virtual, 992K free
```

Using such a layout for SunOS 4.0 was not appropriate, since there was no number available which would make sense when placed inside the parentheses. The current SunOS 4 port displays: available (real) memory, in use, free, locked. Clearly, the machine module needs some control over the labelling of the information. Therefore, not only does it need to pass back the data itself, but also strings describing the data.

### The Design

Some of the decisions made about the module interface were dictated by earlier design decisions pertaining to output display handling. The display engine in *top* does not use *curses* [1] or anything similar to it. As early as version 2, it was decided that *top* could do a better job of optimizing the number of characters output to the screen than any sort of screen or window management software: *top*

compares the numerical data before converting it to ASCII and displaying it on the screen. When the module interface was developed for *top* version 3, the raw numbers were still needed so that the display interface could work pretty much the same way.

But when it came to the process lines, the original version 2 design decision was that comparing the individual numbers did not yield enough of a benefit. For those, the text line was formatted first, then a character-by-character comparison was carried out between the new and old lines, and overstrikes and cursor movement used as necessary to update the screen using the fewest characters possible. This display handler design still exists in version 3. Consequently, the module interface requires that information about an individual process be returned as a preformatted string of text.

### The Function Definitions

Putting it all together, a machine module is expected to have the following functions:

**machine\_init(struct statics \*statics)**  
Carries out any necessary machine-specific initialization. This includes calling *kvm\_init* or similar operations, retrieving values from the kernel that are not expected to change (such as *nproc* and the pointer to the *proc* table), allocating any permanent arrays (such as an array to hold the *proc* table), and doing any other calculations for values that will not change over time. This function also fills in a *struct statics* array with static information: currently arrays of string labels for the first few lines of the display. The structure is documented in Figure 5.

**char \*format\_header(char \*uname\_field)**  
Returns the header line for the process display area. The argument, *uname\_field*, is used as the label for the username/uid column. A command line

---

```
struct statics
{
    char **procstate_names;    process state names
    char **cpustate_names;    cpu state names
    char **memory_names;      memory statistics names
};

struct system_info
{
    int    last_pid;           last process id issued
    double load_avg[NUM_AVERAGES]; load averages
    int    p_total;           total number of processes
    int    p_active;          number of processes active (displayable)
    int    *procstates;       array of process states data
    int    *cpustates;        array of cpu states data
    int    *memory;           array of memory statistics data
};
```

Figure 4: Structures filled in by machine module functions

argument allows the user to choose between user-names and user ids in the display. The machine-independent portion of *top* processes this and decides how the column should be labeled (either `USER-NAME` or `UID`) and passes this as the argument to this function. The function embeds it in an appropriate place in the line that is returned. This function is necessary because the machine module has complete control over the formatting of the individual process status lines. Therefore, it must also have control over the column headings.

`get_system_info(struct system_info *si)`  
Fills in a *system\_info* structure with current information about the status of the system. This is called once per display iteration. The structure is documented in Figure 4.

`caddr_t get_process_info(`  
    `struct system_info *si,`  
    `struct process_select *sel,`  
    `int (*compare)() )`

Retrieves current process information, paying attention only to those processes which meet the selection criteria in *sel*. The information is then sorted by *qsort* (3) using *compare* as the comparison function. It returns an arbitrary value used as a handle for *format\_next\_process*.

`char *format_next_process(`  
    `caddr_t handle,`  
    `char *(*get_userid)() )`

Format and return a string that describes the next process in the sorted list. The first argument is the handle returned by *get\_process\_info*. The second argument is a function that, given a uid returns either a username or a uid (used for formatting the username column).

`int proc_compare(caddr_t p1, caddr_t p2)`  
A *qsort* comparison function suitable for use as the *compare* argument for *get\_process\_info*. It was originally intended that different comparison functions would be made available by the machine module to provide for sorting on different columns of the output. Since the machine module is the only part of *top* that knows how to look in the *proc* structure and that knows how to format a process status line, it would be necessary for the module to provide such comparison functions. Although this flexibility exists in the design, it has not yet been exploited.

`int proc_owner(int pid)`  
Returns the uid of the owner of the process *pid*. This is used in the machine-independent part of *top* to validate the use of the internal *kill* and *renice* commands.

`int setpriority(int dummy,`  
    `int who,`  
    `int niceval)`

On those systems which do not have a *setpriority* (2) system call, this function needs to be provided in the machine module. It is intended to be compatible in a limited manner with the BSD *setpriority* system call. In *top*, the first argument will always be *PRIO\_PROCESS* and can safely be ignored, the second argument is the process id, and the third is the new desired priority. The machine independent part of *top* guarantees that this function will never get called unless the user who ran *top* is either the superuser or the owner of the process whose pid is the second argument. However, the machine independent part of *top* has no easy way to see if the new priority is less than the old priority. It is up to this function to perform that security check (just as

---

```
main()
{
    process_options;
    machine_init(&statics);
    display_init(&statics);
    initialize_signals_and_miscellany;
    while (more_to_display)
    {
        get_system_info(&system_info);
        processes = get_process_info(&system_info, &ps, proc_compare);
        display_load_averages;
        display_time;
        display_procstates;
        display_cpustates;
        display_memory;
        for (i = 0; i < system_info.p_active; i++)
            display_process(format_next_process(processes, get_userid));
    }
}
```

Figure 5: Sketch of main algorithm for *top*

the real *setpriority* call in BSD makes such a check, so must this one). In general, only modules for System V Unix variants will need to supply this function.

The functions defined above enable the machine-independent side of *top* to have a clean structure unencumbered by details about differences in machine specifics. The overall algorithm is given in Figure 5. This hides many of the details, but highlights the use of the machine module functions.

### Labelling Information

The function *machine\_init* fills in a structure with "static" information: data that will not change during execution. This data consists solely of text strings suitable for labelling information returned by *get\_system\_info*. The *statics* structure contains (in the current design) three pointers, each pointing to a NULL-terminated array of strings. In the *system\_info* structure are three corresponding pointers to integer arrays: *procstates*, *cpustates*, *memory*. For each of these array pairs (i.e.: *procstate\_names* and *procstates*), the machine-independent code will display the label after the number. Refer to Figure 4 for the actual structure definitions. As part of its output optimization, the display engine knows that it only needs to write the string labels to the screen once. On subsequent updates it only changes the numerical data, and then only if the data has actually changed.<sup>2</sup> Both process states and memory statistics are handled the same way. If a string in the *\_names* array of the *statics* structure is zero length, then the corresponding element in the array of numbers (from the *system\_info* structure) is skipped. If the statistic is zero, then neither the label nor the number are displayed. The display engine also takes care of pluralization and trailing commas. The statistics for cpu states are handled differently. The numbers in the *cpustates* array are assumed to be in tenths of a percentage point. For example, the integer 105 is displayed as 10.5. Each of these numbers is displayed (with a trailing percent sign) even if it is zero, and they are always formatted to take up 5 columns.

### Important Kernel Data

Although each port deals with differences in kernel structure, methods, and variable names, there are some pieces of information which are commonly essential across most of the platforms.

*proc* The beginning of the *proc* array (an array of *struct proc*).

*nproc* The size of the *proc* array.

*avenrun* The array of load averages. There are almost always three elements: one minute average, 5 minute average, 15 minute average. These are the same numbers shown as "load average" by *uptime* (1). Some systems store this as a float or double while others store it as an integer with an implied binal point.

*mpid* The process id assigned to the last process. Not all platforms assign process id's sequentially: those that don't do not have such a variable.

*cp\_time* An array of counters indicating time spent in each of several different cpu states, typically idle, user mode, niced user mode, kernel (or system) mode. At every clock interrupt one of these counters (depending on the current kernel state) is incremented [3, page 51]. By comparing the counters over time, *top* can calculate percentage time spent in each kernel state. Some System V kernels keep this information in the structure *sysinfo*. The size and significance of the individual members of the array vary as well.

*ncpus* On multiprocessor systems, the number of cpu's installed.

Another important set of information is the statistics on memory usage. But the widely different methods of memory management means that the kernel data structures involved are hardly ever the same between two different platforms. As a case in point, consider the difference between BSD 4.3 for the VAX and SunOS 4. Both were derived from BSD 4.2. On the VAX, all memory usage statistics are collected in *struct vmtotal* as described earlier (see section "The Challenge"). This information can be easily retrieved from the kernel using techniques previously described. Earlier versions of SunOS 4 made a half-hearted attempt to maintain the data in this structure, but the memory management techniques employed under SunOS 4 did not lend themselves well to the gathering of such statistics. As a consequence, the only way to collect meaningful information about physical memory usage under SunOS 4 is by walking the array of page descriptors (*struct page*) in the kernel. This is similar to walking the array of process structures and can be done in one of two ways: either a call to *read* inside a loop (one read per structure) or one large read for the entire array followed by a loop iterating through the returned data. Either way, this is obviously much more involved than retrieving just one structure.

### Conclusions

It is a tremendous challenge to write a program which is so heavily dependent on Unix internals in a way that it is still easy to port. There are many aspects of the machine module design which the author is still not particularly fond of. This design

<sup>2</sup>This isn't strictly true. If the number of digits required changes (i.e., the number to display changes from 9 to 10) then the remainder of the line needs to be rewritten since the text labels no longer appear in the same columns.

works quite well, and those who have engaged in ports to other platforms indicate that it is reasonably easy to use. Some improvements could be made, but in general the design seems to work quite well.

Over the years, *top* itself has become a very popular tool for system administration. The information it provides is invaluable in troubleshooting process-related problems on the system. Several Unix vendors now provide ports of *top* as part of their standard operating system distribution. By any measure *top* can be deemed a success.

New versions of *top* are routinely distributed through the normal free software distribution channels: the newsgroup *comp.sources.unix* and all the sites which archive postings to that group. The latest release can always be found via anonymous FTP at the site *eeecs.nwu.edu* in the directory */pub/top*. The most recent beta test version is usually placed in the same directory. Despite the author's recent change in job status, he hopes that Northwestern University will continue to provide storage for the package on its anonymous FTP server.

#### Acknowledgments

The author would like to thank all the people who have provided suggestions and additional code for *top*, and he would especially like to thank those who took the time to write additional machine modules as well as those who beta tested new versions. Thanks are also extended to Rice University, Northwestern University, and now Argonne National Laboratory for providing facilities and motivation.

#### Author Information

William LeFebvre is a Computer Systems Engineer in the Decision and Information Systems Division of Argonne National Laboratory. He received a Bachelor of Arts degree (with a major in Computer Science) in 1983 and a Master of Science degree in 1987, both from Rice University in Houston, Texas. William can be reached at Argonne National Laboratory, 9700 South Cass Avenue, DIS/900/MS-12, Argonne IL 60439-4812. His electronic mail address is: [lefebvre@dis.anl.gov](mailto:lefebvre@dis.anl.gov).

#### Bibliography

- [1] Arnold, Kenneth, "Screen Updating and Cursor Movement Optimization: A Library Package," *UNIX Programmer's Supplementary Documents*, (PS1), 4.3 Berkeley Software Distribution, April, 1986.
- [2] Bach, Maurice, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
- [3] Leffler, Samuel, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *4.3BSD UNIX Operating System*, Addison-

Wesley Publishing, Reading, Massachusetts, 1989.

- [4] Thompson, K., "UNIX Implementation," *The Bell System Technical Journal*, 57 (6), July 1978.



# Handling Passwords with Security and Reliability in Background Processes

Don Libes – National Institute of Standards and Technology

## ABSTRACT

Traditionally, background automation of interactive processes meant giving up security and reliability. With the advent of software such as Expect for controlling interactive processes, it has become possible to improve reliability and security with relative ease.

This paper reviews the reliability aspects but focuses primarily on the security aspects, presenting several non-obvious techniques for dealing with passwords and other sensitive information in background processes. These techniques require no changes to existing programs and no new security systems are necessary. With the appropriate tools and examples, these techniques can be applied with surprisingly little effort to a wide variety of problems.

## Introduction

Shell scripts cannot automate interactive processes except in the simplest of ways. In particular, data can be written to a process but only following one path through the program. Responding to programs is not possible. Problems such as timing and buffering can make automation difficult if not impossible.

It is possible to reliably automate interactive processes with a variety of tools including C, Perl, and Emacs. For simplicity, I will present examples in Expect [Libes94], but other tools are similarly applicable. Indeed, both Perl's "chat2" [Schwartz90] and the C/C++ Expect library were modelled after the Expect program.

Automating **ftp** is a common problem. The usual solutions are to use an **.ftprc** file or an in-line "<<" script. Unfortunately, these sacrifice both reliability and security. Reliability is lost because these mechanisms offer no way to verify that the commands succeed. Security is lost when passwords are stored as cleartext in a file or passed as cleartext through command-line arguments. (For simplicity, from now on I will refer to all sensitive information as "passwords".) Security systems such as Kerberos [Miller87] do not address these problems.

This paper does not address the simple cases where applications are entirely under your control and can be modified or otherwise forced to run without passwords. **sudo** [Nieusma] and similar programs provide a direct solution to these problems.

In contrast, the problems addressed by this paper demand a password. A simple case might be that of designing a means to use a service from a commercial provider in the background. An automated solution requires you to log in and supply the password. The commercial service is not under your control.

This paper describes several techniques that can be used to handle passwords in background processes in a secure way. The techniques are non-traditional yet relatively simple to implement. These techniques will be demonstrated using Expect.

## Expect – An Overview

Because the examples in this paper are written in Expect, an overview of the language is provided here. The implementation and philosophy of Expect is described at length in the literature [Libes90, Libes91]. Briefly, scripts are written in an interpreted language. Commands are provided to create interactive processes and to read and write their output and input. Expect is named after the specific command which waits for output from a program.

The language of Expect is based on Tcl [Ousterhout94]. Tcl is actually a subroutine library, which becomes embedded into an application and provides language services. The resulting language looks very much like a typical shell language. There are commands to set variables (**set**), control flow (**if**, **for**, **continue**, etc.), and perform the usual math and string operations. Of course, UNIX programs can be called (**exec**). All of these facilities are available to any Tcl application. Tcl is completely described by Ousterhout.

Expect is built alongside of Tcl and provides additional commands. The **spawn** command invokes a UNIX program for interactive use. **send** sends strings to a process. **expect** waits for strings from a process. **expect** supports regular expressions and can wait for multiple strings at the same time, executing a different action for each string. **expect** also understands exceptional conditions such as timeout and end-of-file.

Using Expect it is possible to script **telnet**, **ftp**, **rlogin**, **rz/sz**, and numerous other programs. Many of these tasks fall in the domain of system administration. For example, a system administrator creating thousands of accounts each semester will find an automated **passwd** program much more convenient than having to type in each password manually.

The following script is another example, driving the **fsck** program so that one class of questions is answered "yes" while another is answered "no". If anything else appears, control is temporarily turned over to a user to answer it.

```
while 1 {
  expect {
    eof {break}
    "UNREF FILE*CLEAR\?" {send "y\r"}
    "BAD INODE*FIX\?" {send "n\r"}
    "\\? " {interact +}
  }
}
```

Using a script like this one can substantially raise the reliability of tasks that normally require interactive use.

Expect and related programs can be put to a wide variety of uses as others have found [Woodson91, Morrison92, Stevens92, Caffrey92, Dichter93] solving problems which were not even recognized as problems only because there were no good solutions. A particularly common problem addressed by interaction automation software is entering passwords.

Passwords are usually entered by hand. Most programs (**rlogin**, **crypt**, etc.) use **getpass**, a UNIX library function, which reads the password from **/dev/tty**. Since **/dev/tty** cannot be redirected from the shell, the user must enter keystrokes manually. A variety of kludges have appeared over the years to defeat such security measures. Why? Because entering passwords manually is tiresome. Consider having to enter the same passwords every day to make use of a service.

The remainder of this paper will focus on automating the handling passwords with special regard to background processes. Background processes are a general goal – if you can run a process in the background, it is completely automated. You can turn your attention to other things. This is good.

In many cases, everything in a process can be automated except for the password entry. Were this automated, the process as a whole could be made into a background process. So how do we fix this problem?

In this paper, I will describe several common scenarios involving handling passwords. In each case, I will explain how to automate the handling,

usually resulting in a completely automated and backgroundable process.

I will use the term "*script*" to refer to that which performs the automation and may indirectly run the true "*program*" of interest. Of course, the program may indeed be a script. Similarly, the role of the script may be played by a compiled program. However, the terms I will use are accurate for most applications.

### Technique 1: In the Foreground, Prompt For Passwords Ahead Of Time

The technique described in this section is appropriate for a user who decides at the spur of the moment to schedule a background task for a later time. ("*spur of the moment*" is not meant to imply the command is trivial or light-hearted. Virtually all interactive commands are "*spur of the moment*".) For example, imagine a user wants to automate a **telnet** session to another host. The session must occur several hours later, however. The user will not be present to supply the password.

One solution is to write a script that prompts for the password immediately when the user makes the request. The script begins running interactively. The first thing it does is prompt for passwords. Once all sensitive information has been gathered, the script disconnects from the terminal and continues in the background, perhaps sleeping if necessary until an appropriate time. The script then starts the program, interactively answering the program's requests for passwords.

Below is a sample of such a script using Expect. The script is not setuid and may be readable to others since no passwords are embedded within.

```
# prompt and collect password for later
stty -echo
send "password? "
expect -re "(.*)\n"
send "\n"
set password $expect_out(1,string)

# got the password, now go
# into the background
if {[fork] != 0} exit
disconnect

# now in background, sleep (or wait
# for event, etc)
sleep 3600

# now do something requiring the password
spawn rlogin $host
expect "password:"
send "$password\r"
. . .
```

This script can be extended as necessary. For example, the task might **telnet** to multiple hosts or a additional hosts from the first **telnet**. Each of these in turn requires more passwords. These can be prompted for and collected when the script has begun.

The prompt should make clear what the passwords are for. It may be helpful to explain why the password is needed, or that it is needed for later. Consider the following prompts:

```
send "password for $user1 on $host1: "
send "password for $user2 on $host2: "
send "password for root on hobbes: "
send "encryption key for $user3: "
send "sendmail wizard password: "
```

It is a good idea to force the user to enter the password twice. It may not be possible to authenticate it immediately (for example, the machine it is for may not be up at the moment), but at least the user can lower the probability of the script failing later due to a mistyped password.

```
stty -echo
send "root password: "
expect -re "(.*)\n"
send "\n"
set passwd $expect_out(1,string)
send "Again:"
expect -re "(.*)\n"
send "\n"
if {[string compare $passwd \
    $expect_out(1,string)] != 0} {
    send "mistyped password?"
    exit
}
```

You can even offer to display the password just typed. This is not a security risk as long as the user can decline the offer or can display the password in privacy. Remember that the alternative of passing it as an argument allows anyone to see it if they run **ps** at the right moment.

Even without the **disconnect** command, this is a valuable technique. For example **passmass** is an Expect script that changes passwords on multiple machines simultaneously. This is useful if you have accounts on several machines that do not share password databases yet you want to use the same password on all of them. While this sounds like an obvious security hole, **passmass** can actually increase security. Because **passmass** makes it so much easier to change your passwords on all your accounts, you are much more likely to change them more frequently. And by keeping them the same, you are less likely to have to resort to writing them down in places that you shouldn't. Note that **passmass** is not recommended for widely distributed sites where communications over public networks provides little defense against password exposure. Nor is **pass-**

**mass** recommended for **root**, where this idea is too simplistic and additional precautions should be taken.

### Technique 2: From the Background, Prompt For Passwords When Needed

This technique described in this section is appropriate for commonly occurring tasks such as those that are scheduled at boot time or are regularly scheduled through **cron**.

One solution is to write a simple which runs the program up until it requests a password. The script then tracks down a user (possibly from a list), requests the user talk to it (using **"talk"** or **"write"**). Once connected, the script explains what it wants and why, and then asks the user for a password. The user supplies it, the script disconnects and returns to the background to continue its processing.

In the following example, the script communicates only with a single user. The script uses **kibitz** [Libes93] to communicate. **kibitz** is a **talk**-like program notable in that it allows sharing of a process (e.g., shell) between multiple users. With the **-noprocs** flag, **kibitz** supports communication without a shared process.

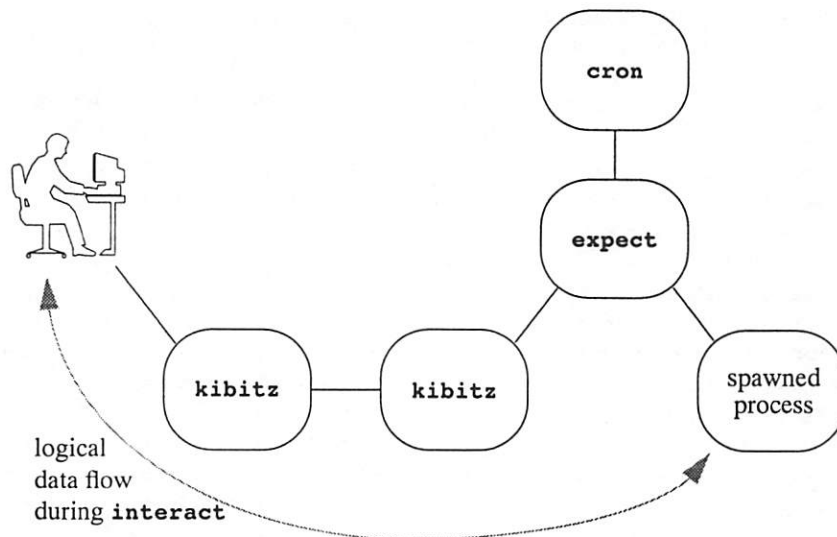
```
spawn kibitz -noprocs $user
```

Once connected, the user can interact with the Expect process or can take direct control of the spawned process. The following Expect fragment, run from **cron**, implements the latter possibility. The variable **proc** is initialized with the spawn id of the errant process while **kibitz** is the currently spawned process. When the user presses the tilde key, control is returned to the script.

```
spawn some-process; set proc $spawn_id
. . .
. . .
# script now has question or problem
# so it contacts user
spawn kibitz -noprocs some-user
interact -u $proc -o ~ {
    close
    wait
    return
}
```

If **proc** refers to a shell, then you can use it to run any UNIX command. You can examine and set the environment variables interactively. You can run your process inside the debugger or while tracing system calls (i.e., under **trace** or **truss**). And this will all be under **cron**. This is also an ideal way of debugging programs that work in the normal environment but fail under **cron**. Figure 1 shows the process relationship created by this bit of scripting.

Those half-dozen lines (above) are a complete, albeit simple, solution. A more professional touch might describe to the user what is going on. For example,



**Figure 1:** Process hierarchy and data flow established when Expect script running a spawned process under cron decides that it needs assistance from a user.

after connecting, the script could send an explanation such as:

```
send "Host frisbee.net is requesting a
      password when I tried to login in
      as user ferdy. Can you tell me
      what the password is (p) or should
      I let you interact (i) or kill me
      (k)?"
```

The script describes the problem and offers the user a choice of possibilities. Here is how the response might be handled:

```
expect {
  k {
    send "ok, I'll kill myself...
          thanks"
    exit
  }
  p {
    send -i $proc [get_password]
    send "thanks!"
  }
  i {
    send "press X to give up control,
          A to abort everything"
    interact -u $proc -o X return A
    exit
    send "ok, thanks for helping.
          I'll take over now"
  }
}
close
wait
# script continues from here
```

This technique can also be used for non-essential information, such as if the script has a question about

what to do in a certain situation, or is performing a backup and needs another tape.

### Technique 3: Protect Cleartext Passwords in Scripts by Permission

The scenario described in the remaining techniques applies when a user does not know a password but needs a service performed that requires the password. For example, mounting devices and initiating backups are typical operations that users need to perform but which require root permission on many hosts.

An obvious solution is to embed the cleartext password in a heavily-protected script. For example:

```
spawn su
expect "Password:"
send "ak3KuIO3\r"
.
```

Schemes to do this without root involvement are well known, such as by using setgid scripts to artificial users and groups created just for the purpose of running such scripts. However, this is difficult to make secure and impossible on some systems when using scripts. Even when using compiled programs, secure handling of passwords is tricky and prone to mishap. The storing of cleartext passwords on a public system is generally a bad idea. There are too many possibilities for lapses of security. These issues are described at length in the literature [Garfinkel91].

This technique is very insecure. Do not use it!

#### Technique 4: Protect Cleartext Passwords in Scripts by Login

It is possible to embed cleartext passwords in scripts and protect the scripts more securely than in the previous technique by placing password-containing scripts on an entirely different host (called the *admin* host from hereon), thereby avoiding file system access holes. Rather than using file system permissions, general shell access is not permitted to the admin host. Instead, each different script is run by logging in to a different account. There are no normal user accounts – only root has access to a general-purpose shell on the admin host.

Writing such a login script to provide a service takes little extra skill than writing any script. Programmers must avoid the obvious pitfalls such as allowing users to invoke a shell or write arbitrary files. However, these are a small subset of the usual concerns in writing setuid scripts. For example, without a shell, users can not change the **IFS** definition or play games with symbolic links.

The key concept here is that scripts can literally store passwords in them with no fear of them being exposed. They cannot be exposed because users cannot read them. They cannot read them because they cannot even log in to the machine in any but extremely restricted ways.

With this technique it is possible to write scripts that log in and connect to other machines which require passwords. For instance, a user may indirectly connect back to their own machine. Imagine a user is working late and wants to suspend the automatic backups that normally run every night at 3AM. The user logs in to the admin host as, say, "**backup-suspend**". The login script for **backup-suspend** logs into the user's host as **root** and suspends the backup. The user might see this interaction:

```
lion% telnet admin-host
login: backup-suspend
Backup suspended on host lion
lion%
```

This particular interaction could be simplified by an email interface since there is nothing interactive here but one might imagine interactions that are much more complex, perhaps even popping up a window on the user's system.

An obvious drawback of this approach is that a second host is required. However, this is not a big deal because computer are cheap. Realistically, most environments have unused computers sitting idle – oftentimes shunned just because they are slow. These slow hosts are entirely suitable for this job since the significant processing occurs on the user's host after the password-containing script has logged in. Although the admin host is executing a script, the admin host is not actually doing the cpu-intensive

work, the admin host is merely telling the user host what to do. The user host is where the significant work is being done.

A second drawback of this technique is that the password is made available for exposure by network sniffing. However, this is a problem for any superuser that logs in over the network.

Finally, it should be obvious that the admin machine must be physically off-limits and its backup tapes must be secure. If either of these are not the case, then obviously the machine is not a safe place to store passwords.

One may draw the analogy that this is akin to placing all of your eggs in one basket. This is quite accurate, however this is a very small basket and easy to keep watch over. Many sites have the analogy of such a basket already, but without realizing or admitting it. Indeed, sites with servers that are kept behind locked doors *are* treating their computers as such baskets.

#### Technique 5: Protect Cleartext Passwords in Scripts by Inetd

In the previous technique, the script is invoked by remotely logging in to another host. An unfortunate attribute of that technique is that some minimal interaction is hard to avoid. In particular, programs such as **telnet** will prompt for the user name. If the user is on a UNIX-like host, they can use **rlogin** which avoids the prompt for the username. If no password is demanded, the invocation is not interactive. This may seem to be a convenience, but is really a necessity when scripts are invoked by other scripts, background processes, or in other situations where the user is not conveniently available to answer the prompts.

For instance, in heterogeneous environments, users can not necessarily depend on the presence of **rlogin**. The **rlogin** program simply is not available from many PCs and Macs for example.

Many programs designed to operate on the heterogeneous Internet stick to the lowest common denominator for communications functionality. For example, Mosaic and Gopher are information systems that follow links of information that may lead from one machine to another.<sup>1</sup> The Gopher daemon does not support the ability to run interactive programs. For instance, suppose you have a **telnet** interface (using the normal **telnetd**) to a valuable resource such as a database. You can make it available through Gopher but only in an uncontrolled way. The Gopher daemon is incapable of running interactive processes itself so it passes the **telnet** information

1. While the Mosaic interface is different than Gopher, both have the same restrictions on handling interactive processes and both can take advantage of the approach described here.

to the Gopher client. Then it is up to the Gopher client to run **telnet** and log in.

This means that the client system has to do something with the account information. By default, the Gopher client displays the information on the screen and asks users to type it back in. Besides being rather silly, it means that accounts and passwords are necessarily exposed to users. Unfortunately, Gopher clients cannot perform interaction automation. And even if they could, the accounts and passwords would still be made available to the Gopher client. By substituting their own Gopher client, users could obtain the passwords and then interact by hand, doing things you (as the advertiser of the service) may not want.

One solution is to use the technique I described in the previous section but modified specifically to run as a **telnet** daemon. **telnet** itself does not demand any account or password, so security is entirely up to the daemon. It is possible to make a non-interactive script simply by not querying for accounts or passwords. A trivial Expect script to run a non-interactive program as a daemon takes no special adaptation. The script merely handles the passwords as before and then runs the program. The client's invocation becomes simply:

```
telnet host service
```

Unfortunately, invocation of interactive programs demands more work because **telnet** clients default to communications with rather peculiar characteristics. Characters are echoed locally and not sent until a carriage-return is entered. Carriage-returns are received by the daemon with a linefeed appended. This peculiar character handling has nothing to do with cooked or raw mode. In fact, there is no terminal interface between **telnet** and **telnetd**.

This translation is a by-product of **telnet** itself. **telnet** uses a special protocol to talk to its daemon. If the daemon does nothing special as in the case of the script that spawned the non-interactive application), **telnet** assumes these peculiar characteristics. Unfortunately, they are inappropriate for most interactive applications. For example, the following Expect script will not work correctly as a daemon:

```
spawn /bin/sh
interact
```

Fortunately, a **telnet** daemon can modify the behavior of **telnet**. A **telnet** client and daemon communicate using an interactive asynchronous protocol. An implementation of a **telnet** daemon in Expect is short and efficient. The basic idea is to make sure that the daemon is always ready to respond to **telnet** commands at all times. This is easily accomplished with an **expect\_before** statement. **expect\_before** provides patterns that are tested before any explicit patterns. Thus, they do not have to be repeated for each **expect** command in an interaction.

A fragment of the Expect dialogue to handle the **telnet** protocol is shown below. Variables such as **IAC** contain the relevant protocol values. The script begins by offering to do echoing instead of the local client. SGA is also offered. SGA (Suppress Go Ahead) means that communication is asynchronous instead of synchronous. The script also offers to support the terminal type.

```
send "$IAC$WILL$ECHO"
send "$IAC$WILL$SGA"
send "$IAC$DO$TTYPE"
```

The **expect\_before** command defines actions for each command that can be sent from the client. For instance, the first pattern matches an acknowledgment by the client that the server should do echoing. The second pattern is similar but for SGA. The third pattern refuses requests from the client to do anything else. The last pattern matches the offer by the client to send the terminal type. In response, the daemon acknowledges by requesting that the client go ahead and send the information.

```
expect_before {
    -re "^$IAC$DO$ECHO" {
        # accept as acknowledgment
        exp_continue
    }
    -re "^$IAC$DO$SGA" {
        # accept as acknowledgment
        exp_continue
    }
    -re "^$IAC$DO\(.)" {
        # refuse anything else
        send_user \
            "$IAC$WONT$expect_out(1,string)"
        exp_continue
    }
    -re "^$IAC$WILL$TTYPE" {
        # respond to acknowledgment
        send_user \
            "$IAC$SB$TTYPE$SEND$IAC$SE"
        exp_continue
    }
}
```

This is not a complete definition to handle the entire **telnet** protocol, but it suffices to give the flavor of it. Indeed, there are near a dozen extensions to **telnet** and more are added frequently. Most **telnet** daemons do not handle most of the **telnet** protocol commands. A richer implementation is shown in [Libes94].

Once the protocol handling is defined, a more typical Expect script can follow. As an example, suppose you want to let people log into another host – such as a commercial service for which you pay real money – and run a single program there but without knowing which host it is or what your account and password are. Then, the server would spawn a **telnet** (or **tip** or whatever) to the other host.

```

log_user 0          ;# turn output off
spawn telnet secrethost
expect "Username:"
send "8234,34234\r"
expect "Password"
send "jellyroll\r"
expect "% "
send "ncic\r"
expect -re "ncic\r\n(.*)"
log_user 1          ;# turn output on
                    ;# send anything that
                    ;# appeared just after
                    ;# command was echoed
send_user "$expect_out(1,string)

```

Additional protocol commands can be exchanged at any time, however in practice, none of the earlier ones will ever reoccur. Thus, they can be removed. The protocol negotiation typically takes place very quickly, so the patterns can be deleted after the first **expect** command that waits for real user data.

```
expect_before -i $user_spawn_id
```

One data transformation that cannot be disabled is that the **telnet** client appends a null character to every return character sent by the user. This can be handled in a number of ways. The following command does it within an **interact** command which is what the script might end with.

```

interact "\r" {
    send "\r"
    expect_user null
}

```

Additional patterns can be added to look for commands or real user data, but this suffices in the common case where the user ends up talking directly to the process on the remote host.

Ultimately, the connection established by the Expect daemon resembles what is shown in figure 2. Notice that the usual **telnet** daemon, **telnetd**, is not part of the figure. Rather, the Expect script plays the role of the daemon. Similarly, the **pty** and the interactive process replace the **pty** and login shell normally allocated and created by the **telnet** daemon.

The daemon could then do any operation involving passwords. For instance, the daemon could **telnet** to yet another host. But in this case the user would

get only what the intermediate server allowed. By controlling the dialogue from the server rather than the client, passwords and other sensitive pieces of information do not have a chance of being exposed. There is no way for the user to get information from the server if the server does not supply it. Another advantage is that the server can do much more sophisticated processing. The server can shape the conversation using all the power of Expect. Without Expect, the user has full access to the spawned interactive program.

In practice, elements of the earlier script (containing the long **expect\_before** definition) can be stored in another file that is sourced as needed. For instance, all of the commands starting with the **telnet** protocol definitions down to the bare **expect** command could be stored in a file (say, **expectd.proto**) and sourced by a number of similar servers.

**xinetd** [Tsirigotis92], a freely-available version of **inetd** provides control on the basis of hosts/networks and time-of-day over access to the services. **xinetd** is strongly recommended over **inetd**.

### Summary and Conclusion

Shell scripts and redirection are so easy to use that users ignore the fact that they provide no reliability or security when it comes to handling passwords in the background. Even users who practice "safe computer sex" in other ways, are negligent when it comes to automation of interactive processes. This paper hopes to enlighten users and save them from the holes into which they will inevitably fall if they stick to the tools and techniques of the past.

The solutions outlined here avoid the historic problems with automating interactive processes in the background. I have shown two techniques that avoid supplying passwords from the command-line (avoiding the well-known "**ps**" hole) or from files (avoiding the "look at the backup tape" and other holes). The remaining techniques store cleartext passwords in files but in such a way that they are inaccessible yet usable by normal users.

Expect-style scripting also offers the ability of reliable control over processes. Scripts can verify responses and can retry or take alternative actions upon failure or unexpected results. When dealing

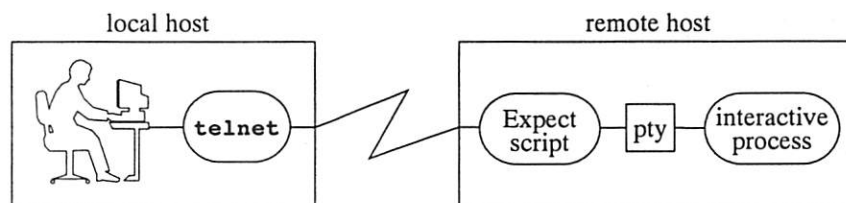


Figure 2: Expect Playing the Role of Telnet Daemon

with users, scripts can also shape the dialogue showing users only parts of the dialogue that are appropriate, or making substitutions in what the user sees.

Expect has been available for several years, yet these techniques are non-intuitive, and for this reason, not known. This paper has shown that each of these techniques requires only a few lines of code with the result that interactive background processes can be automated with security and reliability.

All of the tools mentioned in this paper are freely available and widely portable.

#### Availability

Since the design and implementation of this software was paid for by the U.S. government, it is in the public domain. However, the author and NIST would appreciate credit if this software, documentation, ideas, or portions of them are used.

The scripts and programs described in this document may be ftp'd as `pub/expect/expect.tar.z`<sup>1</sup> from `ftp.cme.nist.gov`. The software will be mailed to you if you send the mail message "`send pub/expect/expect.tar.z`" (without quotes) to `library@cme.nist.gov`.

#### Acknowledgments

Portions of this work were inspired by Sandy Ressler and the NIST Virtual Library Project, and funded by the NIST Scientific and Technical Research Services.

Thanks to W. Richard Stevens, Henry Spencer, Bennett Todd, Miguel Angel Bayona, Brent Welch, Danny Faught, Paul Kinzelman, Barry Johnston, Rob Huebner, Todd Bradfute, Jeff Moore, and Susan Mulroney for providing suggestions that greatly enhanced the readability of this paper.

#### Author Information

Don Libes is a computer scientist at the National Institute of Standards and Technology where he does research related to interaction automation and occasionally logs in as root to "fix things" much to the consternation of the real system administrators there. For the development of Expect, he received the International Communications Association Innovation Award and the Federal 100 Award. He has written over 85 papers and articles as well as two books: *Life With UNIX* (co-author Sandy Ressler, publisher Prentice-Hall) and *Obfuscated C and Other Mysteries* (Wiley). He is presently working on a book called *Exploring Expect: A Tcl-Based Toolkit for Automating Interactive Programs* (O'Reilly). He can be reached at `libes@nist.gov`.

#### References

- [Caffrey92] Paul Caffrey, "User Interfaces and Automating Computer Human Interaction", MSc. Thesis, Amdahl Ireland Ltd., 1992.
- [Dichter93] Carl Dichter, "Surviving Software Testing", *UNIX Review*, pp. 29-36, V11, #2, February 1993.
- [Garfinkel91] Simson Garfinkel and Gene Spafford, *Practical UNIX Security*, O'Reilly & Associates, Inc., June 1991.
- [Libes90] Don Libes, "Expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, pp. 183-192, Anaheim, CA, June 11-15, 1990.
- [Libes91] Don Libes, "Expect: Scripts for Controlling Interactive Programs", *Computing Systems*, pp. 99-126, Vol. 4, No. 2, University of California Press Journals, CA, Spring 1991.
- [Libes93] Don Libes, "Kibitz - Connecting Multiple Interactive Programs Together", *Software - Practice & Experience*, John Wiley & Sons, West Sussex, England, Vol. 23, No. 5, May 1993.
- [Libes94] Don Libes, *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, O'Reilly & Associates, Inc., to appear.
- [Miller87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer, "Section E.2.1: Kerberos Authentication and Authorization System", M.I.T. Project Athena, Cambridge, Massachusetts, December 21, 1987.
- [Morrison92] Brad Morrison & Karl Lehenbauer, "Tcl and Tk: Tools for the System Administrator", *1992 LISA VI Proceedings*, Long Beach, CA October 19-23, 1992.
- [Nieusma] Jeff Nieusma and David Hieb, "sudo" manual page, The Root Group, Boulder, CO, undated.
- [Ousterhout94] John K Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, April 1994.
- [Schwartz90] Randal Schwartz, "Expect.pl", Usenet article id 1990Nov2.003228.22744@iwarpc.intel.com, `comp.lang.perl`, November 2, 1990.
- [Stevens92] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, pp. 635, 653-655, 716, September 1992.
- [Tsirigotis92] Panagiotis Tsirigotis, "xinetd" manual page, University of Colorado, 1992.

1. The ".Z" file is compressed. A ".gz" version is also available which is gzipped.

# Soft: A Software Environment Abstraction Mechanism

*Rémy Evard and Robert Leslie – Northeastern University*

## ABSTRACT

In a traditional UNIX environment, software is installed in many different locations across a distributed filesystem. In order to effectively use the software, users must correctly configure their PATH, MANPATH and other related shell environment variables. A large and dynamic software environment can cause havoc for users as they try to locate programs not in their PATH, resolve filename collisions, and keep up with changes made by administrators, while attempting to update their startup files. In turn, administrators must notify users of new directories or values to put in their files and must spend time debugging users' environments.

A layer of abstraction between the available applications and the way those applications are made available to users through environment variable settings adds a great deal of flexibility for both users and administrators. Administrators can configure and modify software installations without having to notify users of changes. Users may simply indicate which sets of software they desire to use, or they may create arbitrarily complex user-specific modifications. We have implemented this with a mechanism that does not lose speed at login time and which does not use any special shells.

## Introduction

Two years ago, the systems administration staff and computing systems in the College of Computer Science at Northeastern University underwent a significant reorganization. The entire computing environment was rebuilt from scratch, opening a number of opportunities for inventing unique solutions to common administration problems. A complete description of this endeavor, named Tenwen, can be found elsewhere in these proceedings.

One such problem we faced is familiar to many administrators: informing users how to configure their software environment, starting with the PATH environment variable, but continuing with other important variables, such as MANPATH, TEXINPUTS, XAPPLRESDIR, and so forth.

A seemingly simple problem actually has few adequate solutions. The name and syntax of the necessary startup file to be modified depends on the user's login shell, and the actual modifications to make depend on how software is being installed on the system. Neither of these are necessarily something all users would be able to determine for themselves.

An abstraction mechanism simplifies both sides of the user-administrator continuum by separating the details of how, where, and why software resides where it does from the way users can access it: Users can configure their environments easily, reliably, and generally without fuss, while at the same time administrators can be flexible with their choice of software installation design. Changes can be made

to all users' environments simultaneously when necessary to reflect the natural dynamic nature of the computing system.

We have named this software abstraction mechanism "Soft".

## Site Information

The Experimental Systems Group manages the computers in the College of Computer Science, consisting of approximately 350 computers of various types and around 1200 active users. The group is made up of both full-time staff members and student volunteers, totaling an average of 10 people each quarter.

At last count, over 2500 different software packages had been installed on each of our major architectures in the last two years. Our users range from complete novices to expert UNIX programmers.

## Typical Software Installation Factors

The way software is installed greatly influences the way users' environment variables must be configured; there are usually one or more significant directories such as /usr/local/bin where the majority of software is installed. Many other directories are specific to particular applications or computers. Besides the PATH and MANPATH variables, there are sometimes additional environment variables which must be set correctly for certain applications.

All of the above is true regardless of whether a comprehensive method for installing software is used, but having such a method can actually increase

the usefulness of an automatically configured environment. For example, if your software installation method emphasizes the separation of large applications into individual directories, then those directories can be combined and manipulated easily by the administrators in such a way that the applications become available painlessly to users, either automatically, or through a simple modification to one file for those users interested in the applications.

### Problems

The usual mechanism for configuring a new user's environment is to create a default PATH and MANPATH in the user's default startup files. This works well until a new directory needs to be added, and all of the existing users need to be told to update their files. Many will fail to get the message. Inevitably, others will end up with directories in the wrong order for their particular needs, and name conflicts will mystify users until they give in and ask for help. This mechanism also typically fails to address other important environment variables specific to various applications.

Another common mechanism is to set default environments from global shell configuration files, if such global initializations are supported by all of the various shells in use. However, there are often some odd cases which users must handle, and, inevitably, the directories will be in the wrong order for certain people. Also, many people will want to customize the default environment, a feat which will be difficult for them to do without overriding the defaults completely.

The overall result is that users will have to know more about your software installation methods than either they or you want, ending up complicating their lives, constraining your flexibility, and generating unnecessary overhead in terms of endless questions and environment bugs.

### Previous Work

*envv*, by David F. Skoll, as posted to comp.sources.misc, provides a primitive abstraction method by creating shell-independent scripts. However, users must still know which of those scripts to access, and it is expensive at login time.

*user-setup*, from Auburn University (LISA VI paper) allows users to select software packages and creates dotfiles for them. However, it relies on software installations using Modules (LISA V paper), and it only works for csh-style shells, making it virtually useless for many people.

*Depot*, (LISA VI paper) may appear to be related, but in fact addresses a different issue. *Depot* and similar packages manage software installation but do not specifically address user environment issues. Both layers of abstraction are needed in a complex environment.

### Design Goals

We wanted to create a basic abstraction mechanism that allowed maximum flexibility for both the administrators and the users. We wanted to remain in complete control of the abstraction, yet we felt it important that users be able to customize their environment in even the most detailed way possible.

We wanted a system that was fast, and that provided a suitable default environment without any special effort. We also wanted to be able to specialize an environment easily to include specific subsets of software for people who were beta-testing programs, or for people who needed access to administrative programs, and so forth.

Finally, we wanted to be able to group related software in such a way that the entire group could be added to one's PATH in a simple step. This grouping mechanism would be powerful enough to enable users to resolve their own name collisions. For example, users would have the ability to override the standard UNIX utilities with their GNU counterparts if they desired to do so.

### Overview of Soft

Soft works by reading a single file from the user's home directory, *.software*, to determine how the user's environment should be configured. This file has its own simple syntax, independent of any shell.

Each word in this file is an index key to an administrator-controlled database listing all of the directories and environment variables associated with each group of software available on the system.

There are two kinds of database entries: application entries and macro entries. Application entries contain all of the information needed to use a particular application or class of applications: the necessary PATH and MANPATH components, as well as any other needed environment variables and contents. It is possible to create application entries which do not have any PATH or MANPATH components, but are used merely to introduce other variables into the environment.

Macro entries allow several other entries to be combined into a single keyword. A macro may contain references to application entries, or other macros. When a macro keyword is found in a *.software* file, the system expands the macro in a recursive manner according to the entries in the database until it resolves completely into a list of application entries. Macro entries are distinctly identified by a preceding '@' character.

The system processes the *.software* file whenever the user logs in. It creates an executable shell script based on the contents of the file that would customize the user's environment. This script is called the *.software cache* file, because it is

recreated only when necessary. The vast majority of the time, no updating is necessary and the cache file can be executed (sourced) quickly. The cache file must be updated only when either the user's .software file is changed, or when the system-wide database is changed.

### User Interface

We currently give all of our new users a simple, default .software file which gives them our standard supported environment. We have written documentation for the Soft mechanism in the form of man pages, which users can read to learn how to customize their .software file and their environment. The mechanism is heavily customizable, satisfying even the most finicky of our users.

Most users will be content to select from the predefined list of keywords to customize their environment, ordering the keywords in the order they wish to ultimately order their PATH and other variables. At any point, users may insert other special directives into their .software file of the form:

```
VARIABLE=value
```

which introduces an environment variable and an associated value. The two variables PATH and MANPATH are treated in a special manner, such that:

```
PATH=/proj/demeter/bin
```

inserts the directory /proj/demeter/bin into the user's PATH relative to the other path modifiers surrounding it.

For those users who would rather avoid this system completely, they may simply delete their .software file, and no customizations will take place whatsoever. (At our site, several users have done this, but then moved back to using a .software file.)

Since the .software file is shell-independent, users can change shells and have the same software environment without having to modify any startup files. Only the cache file is dependent on what variety of shell is in use. Both sh- and csh-style scripts can be generated, but generating cache files for other types of shells would be a simple addition.

### Implementation

In the following list, \$HOME represents each user's home directory, while \$SOFT represents the Soft installation directory. The complete set of files used by the Soft system is as follows:

- \$HOME/.software: the user's configuration file
- \$HOME/.software-cache.{sh,csh}: the user's shell-specific startup cache files (never modified by hand)
- \$SOFT/software-env.{sh,csh}: the global shell-specific startup files (never change)

- \$SOFT/software-env.db: the global configuration database, where administrators make updates
- \$SOFT/make-sw-cache: a Perl script to create a cache file from a .software file

Installation of the Soft system requires initially creating the software database, and placing hooks into each supported shell's global initialization files to source the appropriate \$SOFT/software-env.\* file. (If a shell does not support any global initialization files, the hook can be placed in the user-level startup files.)

In practice, we have found it helpful to define an alias, "resoft", in each user's environment which will immediately cause any changes in the user's .software file to be reflected in the environment (updating the cache file if necessary). This alias merely invokes the same hook as above to invoke one of the \$SOFT/software-env.\* scripts. It must be an alias, rather than an actual command, so that it can modify the current environment.

Administrators update the database when new directories for software are created, or when a special application requires specific environment variables. Nothing is required when an application is installed in an existing directory and has no special environment variables.

### Experiences

Our user feedback has been completely positive. The number of PATH-related questions we have answered in the last year totals approximately five.

We had to modify the global database quite frequently when it was first built, but have settled down to about once a month or less, depending on the level of activity related to software installations.

The performance of the system has been quite satisfactory. A small amount of time is needed to rebuild the per-user cache file when changes are made, but this is infrequent, and the cost of the entire system is negligible otherwise.

The ability of the system to cope with both sh- and csh-style shells has proven useful not only because it allows users to change shells without grief, but also because it can work simultaneously with windowing startup files (typically Bourne shell scripts), allowing the windowing environment to parallel the user's login shell environment (which may be of the csh variety.) This allows programs invoked from a window manager to be used the same way as programs invoked from the command line.

We have found that the the actual switch from X11R5 to X11R6 as far as users are concerned will be a simple matter of modifying one macro definition in the system-wide database file. In fact,

we plan to move to a completely new software directory structure in the upcoming months without disturbing the user community at all.

### Planned Improvements

There are several ways we expect to be able to improve the Soft mechanism.

We plan to write user-interface software that will help even the most novice users configure their .software files to their satisfaction. The interface will consult our independent software installation system and help database to provide detailed information to the user about the software they are selecting. It will provide an easy mechanism to reorder their PATH, and insert custom directories.

We also plan to extend the .software file format and syntax such that we can conditionalize various pieces only to be effective on particular machines or platforms. For example, if a particular directory only exists on one machine, then it should only appear in users' PATH variable when they login to that machine. Similarly, other variables could be customized in a dynamic fashion depending entirely on the attributes of the host machine. The .software cache files would be constructed with real shell conditionals to achieve this effect.

### Conclusion

The Soft abstraction mechanism has made life much simpler for the administrators and the users of our computer systems. The administrators can change the underlying software architecture and configure software packages without having to notify the user community. The users have a very simple startup mechanism with no loss of functionality.

We highly recommend this system to anyone administering a site of any size.

### Availability

Our implementation of the Soft abstraction mechanism is available via anonymous FTP from <ftp.ccs.neu.edu/pub/sysadmin/soft>.

### Author Information

Rémy Evard has been the leader of the Experimental Systems Group at Northeastern University for two busy years. He received his M.S. in Computer Science from the University of Oregon in 1992, and has worked as a systems administrator and as a consultant for Argonne National Laboratory for the past six years. He may be reached electronically at [remy@ccs.neu.edu](mailto:remy@ccs.neu.edu).

Robert Leslie is a full-time undergraduate student in the College of Computer Science at Northeastern University. He has worked closely with the Systems Group for nearly two years, helping to design and implement many of the changes in the College's computing environment since its

restructuring. He is scheduled to graduate with a B.S. in Computer Science in 1996. He may be reached electronically at [django@ccs.neu.edu](mailto:django@ccs.neu.edu).

### Bibliography

- Colyer, Wallace & Wong, Walter, *Depot: A Tool For Managing Software Environments*, LISA VI, 1992.
- Elling, Ricard & Long, Matthew, *user-setup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves*, LISA VI, 1992.
- Furlani, John L., *Modules: Providing a Flexible User Environment*, LISA V, 1991.

## Appendix A: Sample software-env.db

The following is a sample \$SOFT/software-env.db system-wide database file.

```
#
# Database format:
# - One logical line per entry; physical lines may be crossed using \
# - Comment lines beginning with '#' are ignored
# - Words are separated by whitespace
# - Keys are case-insensitive
# - Ordering in this file is not significant
# - Environment variables may be used freely in path settings
# - Either the PATH or MANPATH may be set to '-' to ignore it
# - No need to enclose environ vars between { }
# - Warning: in case of duplicate keywords, only the last is effective
#
# To update every user's .software cache, touch this file.
#
# (keyword) PATH [ MANPATH [ ENVIRONVAR SETTING ... ] ]
#
#
# System-wide standard settings:
(system)      /usr/ucb:/bin:/usr/bin:/etc:/usr/etc      /usr/man
(sys5)        /usr/5bin
(ccs)         /ccs/bin:/local/bin                        /ccs/man:/local/man
(home)        $HOME/bin/$ARCH:$HOME/bin                  $HOME/man
(dot)         .
#
# Special system/group settings:
(adm)         /ccs/adm/bin:/priv/adm/bin:/local/adm/bin:/local/etc \
              /ccs/adm/man:/priv/adm/man:/local/adm/man
(beta)        /ccs/beta/bin:/local/beta/bin                \
              /ccs/beta/man:/local/beta/man
(sun4-games)  /usr/games
#
# Collections of software:
(GNU)         /local/gnu/bin                              /local/gnu/man
(X11R5)       /local/apps/X11R5/bin                      /local/apps/X11R5/man
(X11R6)       /usr/X11R6/bin:/ccs/X11R6/bin:/local/X11R6/bin \
              /usr/X11R6/man:/ccs/X11R6/man:/local/X11R6/man
(Openwin)     /local/apps/Openwin/bin                    /local/apps/Openwin/man
#
# Application packages:
(TeX)         - - \
              TEXTFORMATS                                \
              ../local/apps/tex/lib/formats:/ccs/apps/tex/lib/formats \
              TEXINPUTS      ../ccs/apps/tex/lib/inputs      \
              TEXPOOL        /ccs/apps/tex/lib                \
              BIBINPUTS      ../ccs/apps/tex/lib/bib
(ObjectCenter) \
              /local/apps/objectcenter/bin:/local/apps/objectcenter/sparc-sunos4/bin \
              /local/apps/objectcenter/man
(ECLiPSe)     /local/apps/eclipse/bin/$ARCH /local/apps/eclipse/man \
              KEGIDIR /local/apps/eclipse
(gcc-2.5)     /local/gnu/apps/gcc-2.5.8/bin /local/gnu/apps/gcc-2.5.8/man
#
```

```

# Research software
(CM)          /usr/cm/bin          /usr/cm/man
(MasPar)      /usr/maspar/bin      /usr/maspar/man
              MP_PATH             /usr/maspar

#
# Miscellaneous
(MH)          /local/apps/mh/bin    /local/apps/mh/man
(NetHack)     /local/apps/nethack/bin /local/apps/nethack/man
(Netrek)      /local/apps/netrek/bin

#
# The following are macros for groups of apps or "latest version".
# They should resolve to a list of other keywords or macros in this database.
(@x-windows)  X11R5
(@games)      NetHack Netrek sun4-games
(@all-apps)   TeX MH

#
# Bundle the essential paths into a macro:
(@base)       ccs system

#
# Choose-your-own-software-bundle ... "home" and "dot" not included:
(@standard)   ccs system GNU @x-windows @all-apps
(@gnu-standard) ccs GNU system @x-windows @all-apps

#
# More bundles, more inclusive:
(@all)        @standard sys5 @games
(@gnu-all)    @gnu-standard sys5 @games

#
# For those users with empty/bad .software, this entry is required:
(@default)    @standard home dot

```

### Appendix B: Sample .software Files

An example of a simple .software file:

```

@base          # Get basic system commands - leave this!
@x-windows     # All basic X utilities
GNU            # GNU software
@all-apps      # All other special applications

```

A more complex .software file:

```

GNU            # Make GNU utils override standard system commands
@base          # Get basic system commands - leave this!
PATH=/lib:/usr/lib # Include /lib stuff
@x-windows     # All basic X utilities
MH             # MH mail system commands

RESEARCH=$HOME/research # Create a custom environ var
PATH=$RESEARCH/bin/$ARCH:$RESEARCH/bin # Include a custom path
MANPATH=$RESEARCH/man    # Custom man pages

home           # Include $HOME/bin/$ARCH + $HOME/bin
TeX            # TeX publishing package
@games         # We like to have fun once in a while...
dot            # Include "." in the path, at the very end

```

### Appendix C: Sample .software Cache File

A sample cache file that was generated from the previous example of a complex .software file:

```
# DO NOT MODIFY THIS FILE DIRECTLY
#
# This file was created automatically by the software system. You can force
# its recreation by altering or touching the following file:
#
# -rw-r--r--  1 django          692 Aug  2 10:16 /home/django/.software
#
# For more information, refer to software(5) and resoft(1).
#
setenv RESEARCH $HOME/research
setenv TEXFORMATS ../local/apps/tex/lib/formats:/ccs/apps/tex/lib/formats
setenv TEXINPUTS ../ccs/apps/tex/lib/inputs
setenv TEXPOOL /ccs/apps/tex/lib
setenv BIBINPUTS ../ccs/apps/tex/lib/bib
#
setenv PATH /local/gnu/bin:/ccs/bin:/local/bin:/usr/ucb:/bin:/usr/bin:/etc:/usr/etc:/lib:/usr/lib:/local/apps/X11/bin:/local/apps/mh/bin:$RESEARCH/bin/${ARCH}:$RESEARCH/bin:$HOME/bin/${ARCH}:$HOME/bin:/local/apps/nethack/bin:/local/apps/netrek/bin:/usr/games:..
#
setenv MANPATH /local/gnu/man:/ccs/man:/local/man:/usr/man:/local/apps/X11/man:/local/apps/mh/man:$RESEARCH/man:$HOME/man:/local/apps/nethack/man
#
# End of cache (v4.4)
```

### Appendix D: Manual Pages

Following are two manual pages, one for the .software file format, and one for the 'resoft' command.

RESOFT(1)

USER COMMANDS

RESOFT(1)

**NAME**

resoft – effect changes in software environment

**DESCRIPTION**

The *resoft* command is specific to the software environment in the College of Computer Science at Northeastern University. It is used to make your current software environment reflect that defined by your *.software* file.

*resoft* is actually defined as an alias; each of the supported shells will automatically define this alias for you. The alias normally does something similar to the following:

```
source /ccs/etc/software-env.csh
```

The actual definition of the alias depends on which shell you use.

**SEE ALSO**

software(5)

## NAME

*.software* – configuration file for user environment

## DESCRIPTION

Your software environment in the College of Computer Science (namely, your *PATH*, *MANPATH*, and possibly other environment variables) is initially determined when you login by the contents of this file in your home directory. The mechanism for doing this is designed to be flexible, easy to use, efficient, and independent of which shell you use.

The file contains a list of words separated by whitespace (spaces, tabs, or newlines). Comments can appear anywhere in the file, following a hash (#) character. Each word can either be a keyword to be expanded by the system, or something of the form

*ENVIRONVAR=value*

which defines an environment variable. The variables *PATH* and *MANPATH* are special cases, and are treated in a special way as described below.

Each keyword in your *.software* file is looked up in a system dictionary to find a translation for your *PATH* and *MANPATH* environment variables. The order of these keywords is important; these variables will be built in the same order that the keywords appear. Some keywords are prefixed with an at-symbol (@); these are used in the same way as other keywords, except the system automatically expands them in macro-like fashion into a list of equivalent keywords without the at-symbol. If any keyword appears more than once (including as part of macro expansion), then the first occurrence of the keyword takes precedence.

As mentioned above, words of the form *ENVIRONVAR=value* can be placed anywhere in your *.software* file to set environment variables automatically for you when you login. The advantage of doing this over setting them in your *.login*, *.profile* or other shell initialization file is that these variables are guaranteed to be set regardless of what shell you use (provided your shell is officially supported by the system). There are two special cases in the way this is handled:

*PATH=/some/bin/directory*

*MANPATH=/some/man/directory*

Each of these directives will **append** the specified directories to the respective variable, not replace its contents. You can freely mix *PATH=* and *MANPATH=* with any other keywords, and thereby easily create a fairly comprehensive environment. You are free to use *\$VAR* syntax to represent the value of an existing variable in these directives.

The exact list of keywords that you can use to configure your environment is determined by a system-wide database. Its contents are subject to change periodically, however the following keywords are generally guaranteed to be valid (case is not significant):

<i>system</i>	The set of system directories containing standard UNIX commands.
<i>ccs</i>	The set of local (CCS) directories, which should almost always precede <i>system</i> .
<i>sys5</i>	The set of System V directories for Sun systems.
<i>GNU</i>	All GNU software and utilities.
<i>X11</i>	The latest revision of the X Window System software (version 11).
<i>TeX</i>	All TeX publishing software, including the proper special environment variables needed to use it correctly.
<i>adm</i>	Directories with special administrative commands, usually only useful to users doing systems administration.
<i>beta</i>	Directories with software undergoing beta-testing. This software is technically

unsupported, but possibly useful or interesting.

*home* The directories \$HOME/bin/\$ARCH and \$HOME/bin (from your home directory).

*dot* The single current directory "."

Unless you have specific needs to arrange the above keywords in a specific manner, the following macros are probably more useful:

*@base* All essential local and system directories (*ccs* and *system*). If you are not using any of the following macros, you should at least use this in your *.software* file, or you may find many common commands missing.

*@standard* This gives you a standard environment that includes essential directories, GNU software, X, and any other applications as defined by the system database.

*@gnu-standard* This the same as *@standard*, except that the GNU software precedes the standard system software, so that certain GNU commands will override their vendor counterparts. This is recommended only if you understand its consequences.

*@all* This is a more general macro that includes a few more things than *@standard*, particularly *sys5* directories.

*@gnu-all* Finally, this is the same as *@all* except that GNU software precedes system software, similar to *@gnu-standard*.

Note that none of the above macros include *home* or *dot* in them, so you should include them yourself wherever you want them. It is not recommended that you use *dot* at all, because it can be a security problem. If you must use it, it is recommended that you put it at the very end of your *.software* file.

The information in your *.software* file is cached. Normally when you login, the system checks to see if this cache needs to be updated. If so, the system rereads your *.software* file to rebuild the cache; if you are using the system interactively, you will see a message informing you of this fact. The system caches your *.software* file into the file *.software-cache.shell* also in your home directory, where *shell* is replaced by either *sh* or *csh* depending on which variant of shell you use.

If you make a change to your *.software* file, you will not see the effect of those changes until the next time you login. However, the command *resoft* can be used to update your environment on-the-fly. If you find for some reason that even after doing this your changes do not seem to be taking effect, your cache file is probably faulty and you can simply delete it and try again.

## BUGS

If there are syntax errors in your *.software* file, the system generally will not try to use it at all, and simply gives you a minimal default environment and a warning message. To discover the problem, review your cache file and look for the error.

If you have no *.software* file at all in your home directory, this entire description does not apply, and the system will not provide you with any default environment. While not strictly a bug, this is not recommended practice. For those users with special needs, however, this may be considered a feature.

## FILES

\$HOME/.software  
\$HOME/.software-cache.{csh,sh}  
/ccs/etc/software-env.db  
/ccs/etc/software-env.{csh,sh}  
/ccs/etc/make-sw-cache

## SEE ALSO

*resoft*(1)

# Beam: A Tool for Flexible Software Update

Thomas Eirich – University of Erlangen-Nürnberg, Germany

## ABSTRACT

Today's workstations often have a limited local disk space. Besides putting the home of the workstation's owner onto the local disk it is reasonable to place frequently used software packages on the disk, too. This reduces network traffic and makes a workstation more independent from file servers. Of course, the replicated software must be kept consistent with the versions on the file servers. This should be done by an automatic update mechanism.

Copying software packages in their entirety would quickly fill up the local disk space. Especially this problem is addressed by *Beam*. Copying the whole software package is merely the simplest form of *Beam*'s update possibilities. A system administrator can rely on powerful features for writing update scripts: merging of several source trees, enhanced file name generation, embedded Perl code, a rich set of update commands which can be arbitrarily combined to form complicated update rules. Additionally, *Beam* has a PACK concept which allows easy adaptation of the update process to the usage pattern of a workstation's owner. To save space on the local disk the user can omit those parts of software packages which are not needed at all (e.g., foreign language user interface) or which are of less interest (e.g., manuals for experienced users). These parts are not missing on the workstation because a symbolic link to the server version is inserted.

## Introduction

The original initiative to develop *Beam* arose from a typical situation in workstation clusters. Workstations often have disks of limited size (300-500MB). Besides putting the homes of the main users on the local disk it would be reasonable to add frequently used software packages. This has the advantage of reducing network load and making workstations more independent from file servers. If a server is down it is more likely that the workstation will survive the downtime without hanging when most of the daily used software is locally available. Furthermore, it is possible to disconnect a workstation from the net and operate it stand-alone because most of the tools with the majority of their functionality have been placed onto the local disk.

Simply copying software packages from servers to workstations is not a good idea. First, the required space would quickly exceed the limits of the local disk. Second, changes made on the server after the copy are not propagated to the clients. Soon, each workstation has different software versions and errors occur because programs do not fit together anymore.

Thus, the idea of *Beam* was born. *Beam* copies software packages and keeps them consistent with the source version. At first glance, this could also be achieved with existing tools like *rdist* [1], but *Beam*'s strength arises from the flexibility and easy customization performing these updates, e.g., several sources can be merged to a single destination.

offers several update commands which modify files while they are updated. There are commands to change ownership of files, to modify the contents of files, to transform symbolic links, and many others. Complicated and powerful update rules can be constructed by composing update commands. Furthermore, most update commands can be customized by Perl [6] code embedded in *Beam* instruction files.

Groups of files and their update rules can be associated with a symbolic name. Such a group is called a PACK. By composing a list of PACK names, the user can easily customize the update process. Customization often requires to exclude parts of some software from update in order to save disk space. These parts are still accessible because symbolic links to the server version are inserted.

In the following chapter we outline the update concept of *Beam*. Then, we present details of *Beam* instruction files and some of the most important update commands. The updating of a cluster of workstations then shows one possible application of *Beam*. Finally, current and related work is discussed.

## The Update Concept of Beam

Software packages are viewed as file trees. The update process of software packages is controlled by an instruction file. This file defines the source path of the software package, the destination path, and associates update rules with path names. The path names are specified relative to the source or destination path. An update rule consists of update commands. Arguments can be supplied to

```

1  $x11 = "X11R5"                if $x11 eq "";
2  $src = "/local.remote/$x11" if $src eq "";

3  sub dst {
4      foreach $d ("/usr/local.stand/$x11", "/local.stand/$x11") {
5          return $d if -d $d; } }

6  sub mainMajor {                # Select for each shared library and each major
7      ... }                      # version the one with the highest minor version

8  sub DPI { ... }               # Find out the dpi of the monitor (75/100)

9  FROM:      $src
10 TO:        &dst
11 NOTIFY:    mail      warning @users
12            file      update  localhost  "/usr/tmp/beam:$src->&dst"
13            syslog    error

14

15 PACK:      std                # standard version
16            **/*.{old,orig}    : delete
17            lib/               : net
18            lib*.{so,sa}.*     : fgen filter=mainMajor : syn
19            X11/
20            X*DB rgb.*         : syn
21            fonts/             : net
22            &{DPI}dpi          : syn
23            misc/              : syn
24            hangl*              : net      # Very big font files
25            jiskan*            : net      # but never used
26            /
27            ...
28            / / /
29            ...

30 PACK:      -misc-fonts        # no misc fonts at all
31            lib/X11/fonts/misc : net

32 PACK:      +misc-fonts        # all misc fonts
33            lib/X11/fonts/misc : syn

34 PACK:      +cc-kit
35            include/X11        : syn
36            bin/{imake,makedepend,xmkmf} : syn
37            lib/*.a            : syn

38 PACK:      light-cc           # smallest stand-alone vers w/ C stuff
39            <std>               : include
40            <-misc-fonts>      : include
41            <+cc-kit>          : include

42 PACK:      subserver          # for a certain architecture (argument: arch=XXX)
43            ...

```

Figure 1: Sketch of an instruction file for X11.

update commands. The following lines depict an outline of a simple instruction file.

```
FROM:  source-path
TO:    destination-path
PACK:  pack-name
p : c1 ... : ... : cn ...
```

One path name *p* is specified in a `PACK pack-name`. It is associated with an update rule consisting of update commands *c1* to *cn*. Suppose the associated update rule is to reproduce the source then the file or subtree *source-path/p* would be copied to *destination-path/p*.

The update process of *Beam* comprises three phases. The first phase parses the instruction file and performs variable substitutions and other preprocessing. In the second phase *Beam* constructs an association of path names with update actions. Path names are subject to file name generation and other mechanisms. The update commands are processed and yield the update action. In the last phase then *Beam* traverses the source and the destination tree simultaneously and performs for each path the registered update action. The update action is executed and creates an internal representation of the destination as it should be. If the actual destination deviates from the internal representation then it is updated. Only the deviating properties are corrected (e.g., only access modes).

Any update rule falls into one of the following four categories. These categories and their effect on a subtree *p* of the software package are explained below. Files, that is, non-directories, are also viewed as subtrees consisting only of a single leaf.

- syn** This class of update rules is the synthesis of the destination subtree. The simplest form of synthesis is to create an exact copy of the subtree *p* and to keep it up-to-date. More complicated synthesis may include the modification of ownership, access modes, or the contents of files. Subsequently, some update commands will be presented which achieve these modifications.
- delete** The subtree *p* is not propagated to the destination. If the destination contains a subtree named *p* it is removed. This class of update rules is useful if certain files shall not be distributed because they contain confidential or private information (e.g., license data).
- net** Instead of copying the subtree *p* to the destination a symbolic link is created referring to the source subtree. Instead of the source path known to *Beam* the user can specify other paths to be used for creating the symbolic links to the network version. This can be useful if the source trees are accessed via a temporary mount point and the net links shall go a different path. This class of update rules is

useful if files shall not consume local disk space but must be still accessible.

**keep** The subtree *p* at the destination side is not updated and remains as it is. Nothing is done if it doesn't exist.

Of course the user can specify update rules for subtrees in *p* pertaining to a different update category than the one registered with *p*. The user can then easily express situations like: link *p* to the server (net) but update *p/q* (syn) and delete *p/q/r*. The directory *p* and the subtree *q* except *p/q/r* are updated. The siblings of *q* become symbolic links referring to the corresponding file in the source tree. The subtree *p/q/r* is not contained in the destination at all. Arbitrary update rules can be nested in any depth. The nesting of update rules allows the user to express update situations in a very compact way. The `PACK std` in Figure 1 depicts repeated nesting of update rules pertaining to different categories.

Up to now we have implicitly assumed that *Beam* updates a software package from a single source. But *Beam* can handle several source trees. A possible application could be to maintain an original software tree and one which contains only customized files. *Beam* can merge them to a single destination tree. The destination is kept consistent with both source trees. There are update commands which control the selection of source files and their processing.

### Beam Instruction Files

All information necessary to perform an update are extracted from an *Beam* instruction file and the arguments passed to *Beam*. An instruction file consists of several sections. Each section starts with a label placed at the beginning of a line. *Beam* parses the contents of sections similarly to a shell. The contents are split into words delimited by whitespace. The quotations '...', "...", and `...` can be used in the same fashion as in a Bourne shell [5].

Additionally, Perl [6] code can be placed in an instruction file before the first section label (see Figure 1 lines 1-8). The Perl variables and subroutines can be referred to from the subsequent sections. Macros for update commands or rules can be written in Perl. Furthermore, many update commands allow the specification of Perl subroutines to adapt their behavior to the user's needs. The instruction file depicted in Figure 1 contains several examples of referred Perl variables and subroutines. For instance, the *dst* subroutine in lines 3/4 computes the correct destination path because it differs among the workstations. Simple problems often can be solved by a few lines of Perl code. The setting of Perl variables also can be controlled from the command line by passing arguments of the form: *var=value*. Figure 2 shows an invocation of *Beam*. The Perl variables *src* and *x11* are set from the command line and override

the default settings established in the embedded Perl code (Figure 1 line 1/2).

In the following the sections of an instruction file and their meaning are explained in more detail:

FROM: the paths to the source trees

TO: the path to the destination tree

NOTIFY: how to notify users about update events; *Beam* offers three possibilities: sending mail, writing a log file, or using the syslog facility. Each of them can be separately enabled or disabled. *Beam* maps update events to the following priority levels. The following priority levels are associated with update events in increasing importance: messages, updates, warnings and errors. Each of the three notification possibilities can be configured to handle only messages of certain priority. Figure 1 shows a sample setup (lines 11-13): a log file contains all update, warning, and error messages; Warnings and errors are also sent by mail and errors are additionally logged by syslog. The subject line of the mail and the log file name can be set up by printf-like format strings.

PACK: a collection of files and their update rules; a section of this type can occur more than once. PACKs are distinguished by a name which immediately follows the section label. Each line is divided by colons into groups of words. The first group describes a set of paths and the remaining ones specify update commands. The first word of a command group identifies the command and the rest of the words are treated as arguments.

The PACK concept contributes crucially to *Beam*'s flexibility. A PACK describes how a set of files is treated during update. The files are usually related to a certain feature of a software package. PACKs can be included from other PACKs. They also can be referenced from the command line. At our department we use the following naming convention: a PACK name starting with a plus character provides for the feature being copied onto the local disk while those starting with a minus prevent the copying. Instead of copying the files related to a feature symbolic links are inserted referring to the server version. Such PACKs are called mixins while all others are basic versions which can be modified by mixins. Customizing of a software package to the needs of a user reduces to composing a list of PACK names. The user can create instant combinations of PACK on the command line as shown in figure 2. The PACKs *std*, *-misc-fonts*, and *+cc-kit* are composed. Often used combinations can be offered in turn as a PACK.

Figure 1 contains a PACK *light-cc* (line 38) defining the same combination of PACK as the instant combination in Figure 2.

In order to facilitate the description of sets of files *Beam* offers a file name generation mechanism and the setting of current working paths. File name generation is a superset of the one available in the csh. The wildcard elements and their meaning are:

- `?`, `*`, `[..]`, `[^..]` from glob and the shells
- `{...}` as known from csh
- `**` matches arbitrary subpaths including the empty subpath
- `^x` evaluates to all file names in the current directory which do not match the pattern *x*. An initial `^`-sign can be matched by `[^]`.

The pattern `**/src/**/^.[ch]` gives an impression of the power of the file name generation. It matches all path names containing a component named *src* and whose last component does not end with `.c` or `.h` (see also Figure 1 line 16 for another example). In rare cases shell like patterns are inconvenient or not powerful enough. In these cases the user can switch to Perl regular expressions. For instance, Sun patch files are easier to describe with Perl regular expression as with a shell pattern: `^+\\.d{6}-d\\d`.

Setting current working paths eliminates the need to write all path names relative to the root directory of the software package. Once a working path has been established path names can be expressed relative to the working path. Working paths can be stacked. They are set up by adding a final slash to a path name and are removed by solitary slashes. If a working path matches more than one directory the subsequent path names are evaluated relative to each of these directories.

### Update Commands

Complex update rules are constructed by combining update commands. For each of the four categories *syn*, *delete*, *net* and *keep* there is an update command with the same name. This command implements the default behavior as described previously. The default behavior of the *syn* command can be modified by prefixing specific update commands. The rest of this section presents some of these update commands. If the user has not prefixed special update commands *syn* will make the destination an exact copy of the source. If the source is a file tree it is traversed and the update rule is applied recursively.

**fgn** controls the details of the file name generation. This command can switch to Perl regular expression. File name generation can be

```
% beam -f X11 x11=X11R6 src=/net/future/X11R6 std -misc-fonts +cc-kit
```

Figure 2: sample invocation of *Beam* referring to the instruction file depicted in Figure 1.

restricted to certain types of files, or to certain file trees. The file pattern in the example below will be viewed as a Perl regular expression. The pattern will be only matched against directories. Additionally, a perl function can be specified which further restricts the evaluated set of file names (see Figure 1 line 18).

```
\w+(\.very)?\.old : fgen regexp type=D : syn
```

**select** is only of interest if there is more than one source tree specified. The default behavior of *Beam* in absence of any explicit selection is to merge the entries of all corresponding source directories. If a file exists in more than one source tree the first one is selected. The order is determined by the order of the source paths in the FROM section. The selection of source files can be restricted to a specific source tree or to a subset of the source trees. The example below illustrates the use of select. The whole subtree *lib* is updated from the first source tree except the libraries *libXYZ\** are taken from second.

```
lib/          : select 1 : syn
libXYZ*      : select 2 : syn
```

**follow** is only active if the source file is a symbolic link. Not the link itself is taken as source but the file it points to. The user can specify a positive or a negative number which means to skip the first n links or to skip n links backward from the end of the link chain. If no argument is given all links are skipped. Additionally, the decision to follow a link can be made dependent on the link contents matching a user specified set of patterns. The example below will skip all those symbolic links as long as they start with either */tmp\_mnt* or */amd*. This prevents the update process to copy the symbolic links which have been created by *amd(8)* or *automount(8)*. Instead the files *behind* these links are copied.

```
* : follow /tmp_mnt /amd : syn
```

**translink** is only active if the selected source file is a symbolic link. The contents of the link are transformed according to the supplied arguments. The user can specify pairs of strings which represent beginnings and replacements. The contents of the link is matched against all beginnings. If a beginning matches it is substituted by the corresponding replacement. The user can name a Perl array and/or pass some pairs as arguments. Additionally, the name of Perl subroutine can be specified which performs even more complex transformations. This command is necessary if a software package contains links into itself or to its environment and the installation path or the environment at the destination site differs from the source site. Example:

```
@Tb = ("/usr/X11", "/local/X11",
        "/usr/local/bin", "/local/bin");
```

```
PACK: ...
```

```
lib man : translink table=Tb /h /home : syn
```

**update** determines whether the contents of a regular file are viewed as up-to-date by three possibilities of checking: comparing modification times, comparing the contents, and calling a user defined shell command. *Update* applies only to regular files. For an example see the next command.

**contents** constructs the contents of regular files in case they are viewed as out-of-date. Usually the contents of the selected source file are just copied to the destination file. If more than one source has been selected all selected files are concatenated and written to the destination. Besides these two built-in facilities the user can specify shell commands to construct the contents. The example below shows the commands *update* and *contents* being customized with shell commands. The usage of a *Beam* macro is also depicted. A macro is essentially nothing different from a Perl subroutine which returns a string. The string is substituted for the macro.

```
MACRO cat+sort {
  local($x)='cat $*|sort -u';
  "update sh='$x | diff - \${DST} : "
  "contents sh='$x >\${DST} : syn"; }
```

```
PACK: ...
```

```
etc/conf : cat+sort
bin/*    : contents sh='cp $1 \${DST}; \
              strip \${DST} : syn
```

**ino** controls the setting of the following attributes stored in the inode: uid, gid, access modes, access time, and modification time. For each of these attributes one of the following operation modes can be chosen: force a certain setting; modify the setting according to some expression (access modes); do not update at all; copy the setting from the source file. The user and group identifications can be either specified by number or by name. The access and modification times can be set to those of some reference file. The example below changes the ownership of all files in the subtree *src* while the group ID is left unchanged.

```
src : ino uid=eirich gid=- : syn
```

*Beam* still has more commands but they are less often needed than the presented ones. Due to brevity, we only list some of the topics of the omitted commands: sharing of files via hard or symbolic links; testing if files are the target of symbolic links; executing shell commands depending on update events which occurred in a subtree; handling device inodes etc. Complex update rules can be constructed by combining these commands. Because commands are sensitive to the file type an update rule may behave differently depending on the type of the processed file.

### An Update Concept for a Cluster of Workstations

An important task in maintaining a cluster of workstations is the keeping of the UNIX installations and replicated software packages on local disks up-to-date. The situation is complicated by the fact that the network is not homogeneous. Often there are several architectures with different kernel architectures. Furthermore, the hardware added to some workstations may also require special software (e.g., a graphic accelerator requires special libraries). Finally, the preferences for software packages may vary from user to user and hence from workstation to workstation (e.g., openwin vs. X11, frame vs. tex).

We have implemented an automated update concept for such a situation using *Beam*. We present the automated update concept as set up in our department. The update includes the UNIX installations of our Sun workstations and several public domain and third party software packages. To simplify the handling of heterogeneity caused by different architectures (sun3, sun4) and different kernel architectures the UNIX installation has been divided into separate trees according to the following criteria:

- machine architecture: this is the main tree. It defines the structure of the UNIX installation and contains most of the software.
- kernel architecture: this tree is related to a specific machine architecture and contains programs and files dependent on a certain

kernel architecture (e.g., *top*, *ps*, *vmunix*, kernel debugger etc.)

- local configuration: this tree is architecture independent and contains files describing the configuration of the workstation cluster. It contains files like */etc/amd.map*, */etc/sendmail.cf* etc.

The file trees introduced above are combined by *Beam* to form the UNIX installation of a certain workstation. These file trees are kept on a file server which is accessible by all workstations. System administrators keep the file trees on the server up-to-date by adding, removing, or patching files. Changes are propagated during the night to the workstations. *Beam* instruction files also exist for all other major software packages.

Simply copying software packages in their entirety would fill up the local disks of workstations very quickly. Therefore, *Beam* instruction files offer PACKs to exclude files related to unused features. These features are still available because symbolic links to the server are inserted but they do not require disk space. Inserting symbolic links has a twofold effect. First, other users than the workstation's owner do not miss parts of any software. They are available but probably not on the local disk. Second, if the workstation is operated off the net only those software is on the local disk which is matched by the usage pattern of the owner. Local disks are not clogged with rarely or never used software. A reasonable stand-alone operation is

```
MAILTO eirich                                # mail occasional output of beam-update to ...
M='mail=i4admin'
#      as-user      instructions  parameters  packs
B      -            SUNOS413      $M          std -sunview -rfs -plot -fortran ...
B      -            X11R5         $M          std +misc-fonts
B      eirich       FRAME         mail=eirich  std +english -french
B      -            BIN           $M          eirich
B      src          LEMACS        mail=eirich  std -bytecomp +vm +perl-mode
B      src          PERL          $M          std -curseperl +include -man
```

Figure 3: A sample instruction file of *beam-update*.

	UNIX installation (sun4m)	X11R5	Openwin	Frame Maker 3.1	Lemacs	Sum of all Packages
Total Package (sun4 only)	101MB	68MB	127MB	32MB	31MB	344MB
Reduced (stand-alone)	41MB	16MB	44MB	18MB	15MB	134MB
Ratio (reduced/total)	41%	23%	35%	56%	48%	39%

Table 1: Comparison of full server versions for sun4 architecture with reduced workstation versions. The workstation can be operated stand-alone with the reduced versions.

possible even with a local disk space of only 300MB. Table 1 shows the effect on the size of a software package if barely used parts are omitted. All packages work properly if the workstation is operated stand-alone. The reduced size is compared to a version of the package which already does not contain files related to other than the workstation's architecture.

Up to this point the local disk of a workstation can be filled and updated but a concept for administration and automation of the updates itself is still missing. The automated update uses the UNIX cron service and the program *beam-update*. A workstation destined for an automated update has to run a cron job calling the program *beam-update*. This program determines the host name of the machine it is running on and reads the file `./.../beam/update/hostname`. `./...` represents the path to the local *Beam* installation. This file specifies the software which shall be *beamed* to the workstation. Figure 3 depicts an instruction file. Each line starting with B describes one *Beam* run. The cron job must be setup under root otherwise it has not enough permissions to update files in the UNIX installation. For certain software packages root permission is not necessary. Therefore *beam-update* allows to specify a user name under who's ID a single *Beam* run is performed. Each *Beam* run is defined by an instruction file, options to *Beam* and a list of PACKS. *Beam* instruction files are implicitly searched in `./.../beam/scripts`. The list of PACKS and the options describe the exact configuration of the software packages on the local disk of a workstation.

All relevant *Beam* files for the automated update are located in the file tree of the *Beam* installation which is shared across all workstations via NFS. Thus, maintaining the update process for all workstations is easy. The only thing that has to be done on the workstation itself is setting up a cron job. The update is performed by the clients and proceeds in parallel. Unfortunately, updates are not possible at all if the *Beam* installation is not available, e.g., because the file server hosting the installation is down. This effect can be turned into a more graceful degradation by updating the *Beam* installation itself to the workstations. In this case only updates for those software packages will fail whose server is down.

### Current Work

The actual version of *Beam* has some limitations which will be overcome by the work which is currently in progress. The current version of *Beam* relies on the UNIX file system to read and write file trees. If software is not available on some network file system update is not possible at all. This is not very restrictive in workstation clusters but limits *Beam*'s general use. Furthermore, NFS access as root sometimes causes problems due to restricted access

privileges. Lastly, if updates of the same software package run simultaneously on several workstations there is some synergetic effect caused by the UNIX buffer cache. Inodes and file data are only read once from disk and are then kept in the cache. But this effect depends on the synchrony of the clients and on the load of the file server host. This synergy could be more efficient if file access to software is controlled by a separate program which does the caching of information relevant to the update process.

The next version of *Beam* will be able to perform updates across a network connection. A daemon *beamd* handles accesses to the file system of the remote host. *Beamd* can be used to read remote software packages (pull file model) or to write updates to a remote host (push file model). Additionally, the new version allows the remote triggering of updates. A software server triggers a client to run an update. The details of the update are completely determined by the client. A configuration file for a *beamd* on each host defines what software is advertised for update and what is allowed to be remotely updated by whom. Different authentication schemes prevent unauthorized access. Even *Beam* instruction files can be read via a connection to a *beamd*. These enhancements are completely transparent to *Beam* instruction files. The software accessible across a network connection to a *beamd* is integrated into *Beam* instruction files by using a special initial path syllable `'/beam'`. The next two syllables describe the host and the software package. This is similar to the `/net` or `/amd` mechanisms with the difference that the `/beam` path syllable is only valid within *Beam* scripts.

### Related Work

There are three other systems that have goals similar to *Beam*: *rdist* [1], *depot* [2, 3] and *track* [4]. *Rdist* uses a push file model while the two others employ a pull file model. Both *rdist* and *track* perform updates across some network connection and do not need a network file system. Hence, they lack the idea of inserting symbolic links to a server version. This fact makes them inappropriate for the presented update situations. They are also quite inflexible because they do not have a mechanism comparable to the PACK concept in *Beam*.

Though the primary intention of *depot* is the management of software environments and not software update some features of *depot* come close to some of the ideas in *Beam*. *Depot* merges file trees by either copying files from some depot or by creating symbolic links into the depot. The number of commands and their combinations are limited compared to *Beam*. Further, *depot* cannot handle operating system files.

### Conclusion

*Beam* has been in use for about one year at our department and it has proven to be very useful. It has been developed because software update tools currently did not match our needs. *Beam*'s power lies in its flexibility. It can operate on several source trees. Features of software packages can be described by PACKs and customization of updates reduces to combining a list of PACK names. *Beam* offers a great number of update commands to control the details of updating files. And finally, the user can add Perl code for further customization of details of update commands and for an individual configuration of instruction files.

### Availability

*Beam* is available via anonymous ftp from `ftp.uni-erlangen.de` as `/pub/beam/beam.tar.gz`.

### Author Information

Thomas Eirich studied computer science at the Friedrich-Alexander University of Erlangen-Nürnberg. He received his masters degree in Computer Science in 1989. Since then he is a PhD student at the department of operating systems IMMD IV. His main research interests are distributed, object-oriented operating systems.

### References

- [1] Cooper, M.A.: *Overhauling Rdist for the '90s*. Proceedings of the Sixth Systems Administration Conference (LISA VI), 1992, pp.175-188
- [2] Colyer, W.; Wong, W.: *Depot: A Tool for Managing Software Environments*. Proceedings of the Sixth Systems Administration Conference (LISA VI), 1992, pp. 151-160
- [3] Manheimer, K., et al.: *The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries*. Proceedings of the Fourth Systems Administration Conference (LISA IV), 1990, pp. 37-46
- [4] Nachbar, D.: *When Network File Systems Aren't Enough: Automatic Software Distribution Revisited*. USENIX Conference Proceedings, Summer 1986, pp. 159-171
- [5] Bourne, S. R.: *The UNIX shell*. AT&T Bell Laboratories Technical Journal, vol. 57, no. 6, part 2, pp. 1971-1990, July-August 1978
- [6] Wall, L.; Schwartz, R. L.: *Programming Perl*. O'ReillyAssociates, Inc., 1990

# Depot-Lite: A Mechanism for Managing Software

John P. Rouillard and Richard B. Martin – University of Massachusetts at Boston

## ABSTRACT

Previous versions of depot type schemes for maintaining software, including the NIST depot[5], Xheir[8], CMU depot[2] and its extensions[10], address the needs of a large site with knowledgeable software specialists. At the University of Massachusetts at Boston, software support is provided by a number of student operators, and faculty members. A new lighter weight depot scheme was needed to reduce the amount of knowledge needed by these less experienced software installers. In addition, we needed a depot scheme that would allow the concurrent operation of multiple versions of the same software package on a given host. This paper presents a refinement of the depot scheme that works well for a smaller site, that does not have dedicated software specialists, as well as a larger site that is interested in getting the efficiency and modularity advantages that the depot scheme offers.

Although we use the *modules* package and the *amd* automounter to ease the administration of depot-lite, neither tool is mandatory for its operation.

## Background

The Department of Mathematics and Computer Science at the University of Massachusetts at Boston provides the computing facilities for 700 students and staff on 4 sparc/10's, one sparc 4/490, and one sun 3/180. The few additional computers (approx. 10 sun3's/386i's and DECstations) are used along with 10 X-terminals and 40-60 z29/vt100 type terminals to access the 6 primary computers.

Because of the diversity of people who use any given computing node, it is very important to be able to provide multiple versions of a software product on each node. It is also very important that all of these versions of software coexist. Previous versions of the depot scheme [2, 5, 8] provided for only one installed version of a software package per host. In our environment, as in many commercial software houses, having access to only a single version of a software package is unworkable. Access to multiple versions of software makes it easy to identify and track bugs between different software versions using "side by side" comparisons of the different software revisions.

The support staff for the Computer Science Department consists of one senior system administrator, one system manager, six student operators at the level of a novice/junior system administrator, and two lab directors. In addition to the paid staff are two volunteers who are responsible for the software setup. One deals with installation and maintenance of utilities from the Free Software Foundation, and TeX. The other does many of the tasks of a senior system administrator, but acts primarily as a toolsmith for the senior system administrator, and the lab directors. In addition to these two volunteer software installers, some of the student operators, as

well as some of the faculty contribute to the installed software base.

## Depot-Lite Introduction

Certain features were needed in depot-lite that we didn't feel were sufficiently well developed and embodied in any of the other depot schemes. Table 1 provides a comparison of what we considered the most important features in choosing a depot scheme. In our environment, the following factors needed to be considered:

- Due to the varying schedules of many classes and research projects, it is necessary to wean people off of older versions of software, while providing the newest version for others. Thus, multiple versions of the same software package must be able to coexist on a given machine.
- In the past changing versions of software occasionally required large amounts of system administration time to track down and resolve problems when were introduced. We needed the depot-lite scheme to support full testing of the software in place. Once the testing phase is over, promoting a software package from the testing to production phase must be simple, and not introduce problems that will prevent proper operation in the production role. Also this promotion should require no knowledge about the particular application that is being promoted. As a corollary to the promotion ability, demoting broken or obsolete software must be an easy task that can be assigned to even a novice student operator, who may be the only person on duty and able to respond to a software malfunction.

- Some of the people who are interested in adding to the available installed software do not want, and in the interests of accountability/security should not have root access, therefore it must be possible to install software without requiring root access.
- Because of the large number of software packages that are available, multiple installers with varying levels of knowledge<sup>1</sup> we needed to minimize the amount of time spent customizing software for the depot system. This leads to two types of depot-lite installations: the collection<sup>2</sup> and the module installations. These two types will be discussed in section "Software Installation".
- Because of the lack of sophistication on the part of some software installers, the directions for depotizing software must be easy to provide and use.
- The exact arrangement of the software (e.g., locations of configuration files vs. sharable data files vs. non-sharable executables vs. sharable executables) must be choosable after installation. Thus the different types of data must all be handed in the context of depot-lite. This allows flexibility in sharing and arranging software by the more experienced software installers, without having to reconfigure and re-install the software package<sup>3</sup>. Once this reorganization has been done,

<sup>1</sup>Some don't even know about make(1).

<sup>2</sup>The use of the term collection should not be confused with the CMU use of the term collection. In the CMU depot paper, our collection is termed an environment.

<sup>3</sup>Note that this may require the use of symbolic links, however configuration files are usually read once at software startup, and as such don't generate much traffic.

the installed product can be used as a template by novice/junior software installers when they install the next iteration of the software.

- Although we support mainly computer science students, who should be able to maintain and manage their own environments, we also have a number of people who are interested in using the computer as a tool rather than as a learning exercise. Because of this, there must be easy access to the new tools.

The depot-lite that developed produced the following attributes:

- Supporting multiple versions of the same software is trivial. Full application testing can be done in place. Promoting software to production level, or demoting old software can be easily done with a one line change in an automount map.
- Tracking down all of the parts of the application is easy. Since all of the files needed to run the application are separated into a tree hierarchy with well known names, moving, replicating or sharing or removing the entire installation is greatly simplified.
- Software installation responsibility can be delegated to people without root access, or without general access to the /usr/local tree. As a matter of policy we try to reduce the need to use the superuser account as much as possible. Using depot-lite, the only steps that need to be done as root are:
  - Create the root directory for the tools, and change the owner of the directory to be the person building/installing the tool.
  - Add an entry to the /tools automount map to point to the newly created directory.

Feature	Depot Lite	NIST Depot	CMU	CMU Ext	Ericsson	Xheir
Supports multiple versions	Y	N	N	N	Y	N
Software testable in place	Y	Y†	Y†	Y†	Y	Y†
Easy transition from testing to production	Y	N	Y‡	Y‡	nd	Y‡
Large learning curve for system	N	Y	Y	Y	na	Y
Requires custom tool installation	N	Y	Y	Y	na	Y
Software installable by non-root user	Y	Y	N	N	na	Y
Software transferable between collections	Y	na	N	N	na	na
Supports per domain customization	Y	Y	N	Y	na	Y
Easy access to tools	Y	N	Y	Y	nd	Y
Heavy reliance on symbolic links	N	N	N	N	nd	Y
Has self-contained trees	Y	Y	Y	Y	Y	N

† Testing requires an entire host dedicated to testing

‡ Automated tools are required

• na/nd Not Applicable/Not Discussed

**Table 1:** Comparison of main features found in various depot schemes.

- Optionally a module file can be created that is owned by the software installer. Once these steps are done, the person working on the tool is free to do the installation as a regular user. No special group or user permissions are required.
- Very low learning curve. The installation of the tool under depot-lite does not require large amounts of expertise for a particular tool. The stock installation procedures for the tool can be used to a large extent thus reducing the lead time needed to provide new tools.
- Reduces the chances of inadvertently modifying the software installation. Once newly installed software tree has been unchanged for 3 weeks, an automatic script changes the owner of the files and directories, and removes write permissions from the files and directories in the tree. Once the software is locked, there is no way for the original installer to change it without root access.
- Saving disk space is automated. We have a utility program scans the installed software tree and links together files that have identical permissions, owner, group and contents. In addition putting shareable files into a particular location allows an automated procedure to replace real files with links to this special area.
- Categorizing files into shareable, administrative etc. can be deferred until later with a minimal penalty.
- Depot-lite attempts to minimize the number of symbolic links that are dereferenced over NFS in a software installation. This is particularly important since NFS doesn't cache files as does AFS.

### Server Filesystem Layout

The server view of depot-lite differs from the "classic" NIST Depot [5] in that files are grouped

by architecture and OS before they are grouped by software package. Figure 1 displays this difference in detail.

Since the architecture split occurs first, it is possible, with a single mount, to provide a fully functioning `/tools` hierarchy to a workstation. Tracking the software revisions available on the various platforms is also made easier by splitting on the architecture first. The ability to see every version of software available for a given platform, in a tools hierarchy by performing an `ls` is a large win when developing automated software auditing tools.

We set up the names of the architecture and OS directories to correspond to the identifiers that AMD[6] uses for its architecture and OS version. This allows us to easily create a generic map entry for any given software product. Appendix A shows a sample generic entry for our tools map. In addition to each of our standard architecture directories (e.g., `sun4-sos4`, `mips-u4_2` (Ultrix 4.2)) we have two special directories: `share` and `domain`. The structure of the `share` and `domain` directories is the same as the structure of the architecture directories. Each software installation has its own directory tree hierarchy with a name like `share/xv-3.00a-11`. The items in the `share` tree are shareable across multiple architectures. There are two types of sharable trees: `installation` and `skeleton`.

Installation trees are created for tools whose installation hierarchy supports multiple architectures and platforms by default (e.g., Interleaf). Other tools that can be made shareable because the tool is primarily script based (e.g., `modules`, `majordomo`) also form installation trees. Since the installation trees have all of the necessary files that implement the software tool, these trees can be copied and will provide the complete software package.

Skeleton trees are used to minimize the disk space needed when replicating installation tree hierarchies. A skeleton tree is created by copying an

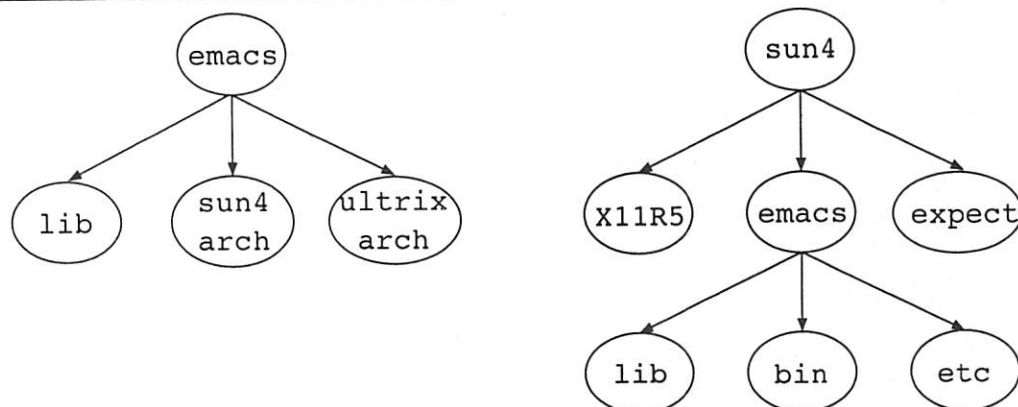


Figure 1: A view of the NIST and depot-lite arch splits. The figure on the left is the NIST server layout, while the view on the right is the depot-lite server layout.

installation tree from one of the architecture specific directories, and then removing all files that are architecture specific. This leaves a skeleton tree of files that are shareable between platforms. Manual pages and documentation are ideal candidates for the share tree since applications are rarely dependent on them for proper operation. Files that are shareable between architectures, but are required for proper operation can also be placed into the share tree. This share hierarchy can then be replicated on another server to provide a measure of fault tolerance. Unlike an installation tree, a skeleton tree is not a viable or usable tree. Some of the files necessary for the software packages operation are missing. The sole purpose of the share tree is to reduce the size of the installation tree so that the installation tree can be replicated utilizing less disk space.

The domain tree is similar to a skeleton share tree. However, it contains only configuration files that are likely to change from domain to domain. This permits different domains to maintain their own directories that are referenced via symbolic links from the installation tree. Using this mechanism each domain or department can customize the installed software. This mechanism could be used on a per machine basis if necessary.

### Operations on Software in Depot-Lite

Software in depot-lite goes through the usual stages: installation, test, lockdown (or freeze), production, and deletion. There are two methods for installation depending on what method will be used to allow users to access the software. Testing is done "in place" making it much less likely that errors due to changing environments will go undiscovered. Locking or freezing the installed distribution attempts to provide a stable version that can not be modified without leaving some record of the change. Promotion to production status is easily done by even the most inexperienced operator, and may occur before or after the software is locked, although locking before promotion is the preferred order. Deletion of software is a multi-step procedure that provides for a test interval to make sure that the software is indeed removable.

### Software Installation

Because we wanted to minimize the time required to build and install a software package, we decided to try to go along with the software provider's installation/build process as much as is possible. To accomplish this, we created two different installation models: collection and modules.

A collection installation requires that the installed software conform to a standard tree format with bin, man, lib, doc, etc and include subdirectories. It can be used for many tools that expect to live in /usr/local. Most of the gnu utilities that

use configure<sup>4</sup> can be installed easily using a collection installation.

The modules installation is easier to use for a large software package that has a lot of expectations about its installation environment. In addition software that is not in general use<sup>5</sup> is often handled as a modules installation. A module installation does not require major restructuring of the makefiles and code to impose some external structure on the authors installation tree. Users access this software by changing their execution search path as in the "classic" depot[5]. At UMB we have automated this operation through the use of modules[3]. Also we use modules as a discovery and documentation mechanism for these tools.

For either installation type the root of the installation tree is changed to its canonical location:

```
/tools/<software name>-  
  <software version>-<local modifier>
```

where software name is the name of the software e.g., xv, emacs, perl etc. The software version is the release of the software in a dotted version form, e.g., 2.14pl11, 3.00a, 4.036 etc. The local modifier is used when the installed version of the software supports features that were added locally. For example, some of the initialization code of xv version 3.00a was rewritten to allow setting of the title and the resource name that would be used by xv. Since this was the first modification that was done a local modifier of -l1 was used, and this version of the software was installed into /tools/xv-3.00a-l1.

### Installation for Modules

Once the root of the installation tree is changed, the rest of the software can be installed exactly as the author of the software intended it to be installed.

The ability to use the author's makefile setup and installation instructions can greatly reduce the amount of work needed to depotize complex tools as compared with the approaches of the CMU depot[2], NIST depot[5], or Xheir[8].

### Installation for Collections

Besides the change that implements a different root directory, using the automatic collection creation utilities requires that the software form a standard Unix tree with bin, etc, lib, man and other directories. If the software is set up to install into a directory such as /usr/local, then this hierarchy of directories is already provided by the author of the software, and no additional work is needed.

<sup>4</sup>The --prefix option makes changing the root of the install tree trivial.

<sup>5</sup>For example PBM utilities, Tex/LaTeX, system administration utilities

If however, this structure is not provided, extensive work may have to be done to change the software's idea of its installation environment. In cases like this, a modules installation for the software is preferable.

### Software Testing

Testing of newly installed software is done by prepending the path of the newly installed software package to the users PATH environment variable. Then the user is using the new version of the software rather than the older version. Once the tool has been sufficiently tested, it can be installed as a default version by simply remapping the simple name<sup>6</sup> to the new version name e.g., (emacs → emacs-19.33) by changing a single line in the auto-mount map.

### Locking the Installed Software

On a daily basis a script is run across all known tools trees on each server. This script performs the following actions:

1. See if the tooldir has remained unchanged for 21 or more days.
2. If so, remove write permissions from all files and directories in the tooldir.
3. Save the name of the owner in the file .owner in the root of the tooldir.
4. Change the owner of the tools to root and the group to daemon making special note not to change any suid/sgid executables.
5. Print a list of the locked tooldirs that have been found for use by the script that links identical files together to save space.

Once the locking is done, the files are unwritable by the original installer. Any changes that are made after the "stabilization period" of 21 days need root access. If changes have to be made after lockdown, the person who wants to make the change runs a script that breaks any hard links that were created, makes the file writable, and changes the group and owner to those of the user invoking the script. In addition to changing the files, it sends mail reporting that a late change is being made<sup>7</sup>.

### Deleting Old Software

Deleting old software is a multi-step process that is carried out by a single script.

1. Notify all users that a tool will be going away via mail to msgs, and an announcement in the appropriate newsgroups.
2. Deny access to the the top level directory for the tool by changing the directory mode to 000. This will cause any attempts to access the files under the tree to fail, and makes backing out a deletion trivial. Simply changing the permissions on the directory back to

<sup>6</sup>The simple name is merely the name of the package with no version information.

<sup>7</sup>Note that the unlocking script has not yet been written.

555 will make it available again, and will also cause the automatic deletion process to abort.

3. Schedule an at(1) job three weeks in the future that will perform the actual deletion using `rm -rf`. Also scheduled at the same time are jobs that mail reminders to the system administrators that the tool is going away.

### Depot-Lite Operations

A number of scripts and techniques have been developed to support maintenance operations in depot-lite.

### Creating Collections

This is the one part of depot-lite that requires a bit more hand operation. In previous attempts to solve this problem, solutions were based on cp and autoexecuted shell scripts and lfu<sup>8</sup> [1]. However the complexity of specifying the files to be included was getting prohibitive, and was going away from the "lite" philosophy. We now create a master collection by hand using some tools that automate the process of linking the bin and man subdirectories for a tool. The collection is automatically propagated to various machines.

Since the master tree is hand crafted, modules type installations can have their binaries placed into collections, which is an advantage. Also, we have discovered that the collections are not often modified, the contents of the files that comprise the collection change as newer version of software are developed, but the tools that comprise the collection don't change often. The number of links that have to be created are small since only the man and bin directories<sup>9</sup> have to be linked into the collection.

The propagation of the master directory is done by a modified version of the nightly program written by Hal Pomeranz[7]. Nightly was originally written to mirror one tree hierarchy, using symbolic links, into an area such as /usr/local on a local workstation. It then would keep track of file accesses in the /usr/local tree, and replace frequently accessed files with real copies of the files rather than just symbolic links. Using nightly we get local file caching that reduces the NFS traffic across our network, and it also provides a measure of fault tolerance since heavily used programs are cached on local disk and are thus not subject to the whims of NFS service.

Before putting nightly into production however, we had to "cache" the symbolic links in the master file system. Given the following setup:

<sup>8</sup>The version of lfu was modified to allow forming a collection from discrete trees.

<sup>9</sup>The other directories in the collection: lib, etc, doc and so forth are not used by the software, and users prefer to run the /tools tree to locate library and other files if they have a need to do so.

- The master filesystem is located in `/tools/local`
- The file `/tools/local/bin/expect` is a symbolic link to `/tools/expect/bin/expect`

nightly would make the local image of the file, `/usr/local/bin/expect`, a symbolic link to `/tools/local/bin/expect`. When executing the file this extra link causes an unnecessary NFS readlink. So nightly was modified to cache symbolic links. When nightly runs in link caching mode, the local image of the file, `/usr/local/bin/expect`, would be a symbolic link to `/tools/expect/bin/expect`. This eliminates an NFS readlink, but still allows the local caching ability of nightly to function.

### Providing a Default or Collection Specific Version

We have found it useful to provide an entry in our automounter maps that refers to the default version of the software that is installed. This entry has the form `/tools/<software_name>`. This is used by scripts (e.g., `expect`, `tk`, and `perl`) so that magic numbers such as `#!/tools/tk/bin/tk` keep working even if the version of the software changes.

Keeping all of the collections pointing to the proper versions of the tools may seem to be a problem especially when multiple collections need to be supported (e.g., `/usr/old`, `/usr/new`). We handle this by adding symbolic links based on the name of the software (e.g., `/tools/emacs.new`, `/tools/emacs.old`). The software in the collection is then accessed through these links. Thus changing the software that is part of the old and new collections is made much easier.

Because we support multiple versions of some programs that have recently changed major version numbers (e.g., `expect`, `tk`, `emacs`) we have found it useful to provide "base version" entries such as `emacs-19` and `perl-4`. This way scripts that work under a previous major revision have a simple way of getting access to the default installed version of the earlier revision of the tool.

### Conserving Disk Space

The primary mechanism for conserving disk space is an automated script that finds files with identical permissions, owner, groups<sup>10</sup> and contents (the discovery step) and hard links them (the linking step) together. If your server exports tools repositories for multiple architectures, the savings can be considerable.

If however, you split each architecture onto different partitions, or onto different hosts, another mechanism for saving disk space is needed. At this

time the discovery mechanism has not been automated, although the "linking" step has been.

In order to use the linking script, the directory `/tools/share/<tooldir>` is populated with the tools tree stripped of all architecture dependent files. After populating the share directory, a script is run on each server that compares the files in `/tools/share/<tooldir>` against the equivalent directory in the architecture dependent tree. All files in the architecture dependent tree that are identical to files in the share tree and are not multiply linked are replaced with symbolic links to their shared counterparts. Again in normal operation, only locked software trees are candidates for this operation.

The reason that the link count is looked at is to eliminate unnecessary symbolic links. It is assumed that each disk that has a skeleton share tree, can also have a corresponding full install tree. Since there is no space to be saved by replacing a hard link between the shared and install tree with an equivalent symbolic link, files that have more than one hard link to them are not candidates for replacement by symbolic links. Excepting files of this sort from being symbolically linked saves some disk space, but more importantly unnecessary NFS readlink calls are avoided.

### Providing per Division/Domain Configuration Files

A technique similar to the `/tools/shared` tree is utilized to maintain configuration files that may be different between different departments that utilize a shared tools repository. The directory `/tools/domain/<tooldir>` is searched, and symbolic links to the files in this directory are put into the architecture dependent tree.

To use this all departments that utilize the shared software tree must have their own versions of the configuration files. Some automatic discovery can be provided by scanning the automount tree looking for unsatisfied symbolic links into the `/tools/domain` tree, but an automatic procedure to distribute the configuration files is lacking. This makes it very important to lock down the exported software tree so that you don't inadvertently break another department's installation.

It would be very nice to have an automatic mechanism for distributing configuration files to other departments when the need to split off a configuration file develops. One scheme that has been proposed is to keep a copy of the configuration file in the shared tree so that the remote department can pick up the default configuration file from the `/tools/share/<tooldir>/domain` hierarchy and install it into their own domain hierarchy.

<sup>10</sup>This forces the conservation step to take place after the software is stable and has been locked down.

### Finding Software Interdependencies

One problem with many tools, especially library packages such as tcl, or perl 5 is that the library may be removed but tools using that package aren't removed.

To discover such problems, a script scans all tool area files on a server for the string `/tools/`. If the string is found, it is truncated to the name of the root tool directory. For example if the string is `/tools/tk-3.3/lib/tk`, the path that is used is `/tools/tk-3.3`. The script checks for the existence of the file `.owner` in the tool's root directory (e.g., `/tools/tk-3.3`). If that file is not found, the missing tool directory has its permissions changed so that it is again available. In addition mail is sent reporting that the tool was "recovered". If the tool can't be recovered (because its tree was really deleted), mail is sent warning that the tool has been removed, but it should still be active.

There is one complication with this setup. As mentioned in section "Deleting Old Software", the first step in deleting old software is to change the directory permissions to 000. Some operating systems allow a program running as root, to find the file `.owner` even though access is technically denied. To combat this problem if you are so afflicted, the script can be run as a non-privileged user. Sadly the need to run as a non-privileged user also prevents the script from automatically reactivating the directory by changing it to mode 555. Consequently a warning is sent to the appropriate parties.

### Scanning Automount Tables

One problem we haven't dealt with yet is the duplication of information concerning what tools are available. This is stored in two locations: the filesystem and the automount map. Tools get deleted or moved and as we all know storing information in two places provides the opportunity for the data to get out of sync.

To combat this problem, all entries in the automounter table are scanned looking for the file `.owner` in every directory under the `/tools` automount point. If the automount point under `/tools` exists, but the file is missing, then mail is sent to warn that there may be a mismatch between the advertised tools, and the available tools.

### Using Depot-Lite Without an Automounter

At UMB we use the amd automounter to maintain our maps. A similar setup can be done with a Sun derived automounter using the ability to create a map with variables that are expanded at runtime to select the architecture and operating system of the mount. However a properly arranged depot-lite can work well without the automounter.

Both methods for using depot-lite without the automounter require symbolic links to map the real mount location for the various tools to their logical

location under `/tools`. The first scheme introduces a performance penalty since the symbolic links are being accessed by NFS.

Create a directory that has symbolic links for every package pointing to a mount area contained in the `.mount` subdirectory. Mount this directory on the client system as `/tools`. Then mount all other tool areas<sup>11</sup> into the `/tools/.mount` subdirectory.

The second mechanism is the same as the first with the exception that `/tools` and the symbolic links are all on the local disk of the workstation thus eliminating the initial symbolic link lookup via NFS.

In the second case, it is easy to create a tool that scans all the mounted directories and updates/creates new symbolic links in `/tools` as needed. It is possible to create a shell script that automatically repoints the symbolic links in response to a downed server thus providing a minimal level of redundancy.

### Conclusion

In response to some specific circumstances at our site, we have developed a depot like mechanism that embodies a number of features that we found critical to smooth operation:

- Depot-lite allows the concurrent operation of multiple version of the same software.
- Depot-lite achieves all of the regular benefits of software modularization without requiring an intricate and extensive series of supporting programs.
- Depotizing a software package under Depot-lite does not usually require a redesign of the installation procedure.
- Depot-lite supports multi-domain sharing of software packages through a flexible yet deterministic file tree layout.

All of our work is not yet done however. Better tools to assist in creating collections need to be developed.

The current interdependency tool calls `strings(1)` directly from within a perl script, however, `strings(1)` sometimes misses valid strings. It would be much better to write a strings replacement in perl and use that rather than the `strings(1)` program.

Also the timing interdependencies of the various checking tools need work. We receive a number of spurious warnings about tools not being present when they are in their three week waiting period to be deleted.

Despite these shortcomings in our automated tool suite, we consider depot-lite useful.

<sup>11</sup>Obviously, the fewer the tool areas (not tools) that are present, the easier this scheme is to implement.

### Availability

The tools and slides will be available via anonymous ftp from `ftp.cs.umb.edu` in:

`/pub/bblisa/talks/depot-lite-tools.tar.Z`  
`/pub/bblisa/talks/depot-lite-slides.tar.Z`.

### Author Information

John Rouillard graduated from the University of Massachusetts at Boston with a B.S. in Physics in 1990. Since then he has been a contractor specializing in tool building and automation of various system administration tasks. In January of 1994 he took over release engineering and development from Brent Chapman for the Majordomo mailing list management tool. At the same time, he became a Senior Systems Consultant for the Mathematics and Computer Sciences Department's Software Engineering and Research Labs at the University of Massachusetts at Boston where he continues his system administration automation tasks for various clients of the Lab.

Richard Martin attended Merrimack College and UMass-Boston. He has been System Programmer with the department of Math and Computer Science since 1984, maintaining hardware and software and training sysadmins.

### Bibliography

- [1] Anderson, Paul, "Managing Program Binaries in a Heterogeneous Unix Network", LISA V proceedings, September 1991, pp 1-9.
- [2] Colyer, Wallace and Walter Wong, "Depot: A Tool for Managing Software Environments", LISA VI proceedings, October 1992, pp 153-159.
- [3] Furlani, John L., "Modules: Providing a Flexible User Environment", LISA V proceedings, September 1991, pp 141-152.
- [4] Lundgren, Thomas, "A File Server Directory Tree Hierarchy", Based on Ericsson Telecom Document ETX/TX/DD-91:110 Uen Rev. B, January 16, 1992.
- [5] Manheimer, Kenneth, et al., "The Depot: A Framework for Sharing Software Installation Across Organizational and Unix Platform Boundaries", LISA IV proceedings, October 1990, pp 37-46.
- [6] Pendry, Jan-Simon and Nick Williams, "Amd The 4.4 BSD Automounter Reference Manual", Supplied with the amd software, March 1991 for version 5.3 alpha.
- [7] Pomeranz, Hal, "A Simple Caching Strategy for Third-Party Applications", SANS III Proceedings, USENIX Association, 1994, pp. 117-122.
- [8] Sellens, John, "Software Maintenance in a Campus Environment: The Xheir Approach", LISA V proceedings, September 1991, pp 21-28.
- [9] Smallwood, Kevin C., "Guidelines and Tools for Software Maintenance in a Production Environment", LISA IV proceedings, October 1990, pp 47-70.
- [10] Wong, Walter C., "Local Disk Depot - Customizing the Software Environment", LISA VII proceedings, November 1993, pp 51-55.

### Appendix A: Generic Tools Automount Entry

Figure 2 shows a typical map entry and the default options for the tools map. The `/defaults`

---

```

/defaults          type:=nfs;opts="ro,intr,nodev,grpidd"; \
                   sublink:="/key}"

emacs  type:=link;fs:=/tools/emacs-19.22;sublink:=.
emacs-19  type:=link;fs:=/tools/emacs-19.22;sublink:=.
emacs-19.22  -type:=link \
             host==terminus;fs:=/disk/sd1f/tools/${arch}-${os} \
             host==lab;fs:=/usr/local/tools/${arch}-${os} \
             || \
             -type:=nfs;opts="rw,intr,nodev,grpidd" \
             host==cs;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os} \
             host==apollo;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os} \
             || \
             -type:=nfs \
             arch==sun4;rhost:=terminus;rfs:=/usr/local/tools/${arch}-${os} \
             arch==sun4;rhost:=ra;rfs:=/usr/local/tools/${arch}-${os} \
             arch==sun3;rhost:=lab;rfs:=/usr/local/tools/${arch}-${os} \
             os==u4_2;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os} \
             arch==sun386;rhost:=ra;rfs:=/disk/id000h/tools/${arch}-${os}

```

Figure 2: The standard amd automounter map entry for a tool.

line specifies the default options that apply to the automounted file system unless overridden. We assume that all mounts are of type `nfs`, and the `nfs` mount uses the options: `readonly`, `interruptable`, no devices, and `bsd` group semantics. The `sublink` option specifies that the automount directory link (e.g., `/tools/emacs-19.22`) should point to a subdirectory of the mount directory specified by the key (i.e. `emacs-19.22`). This is needed since the mount is done at the architecture/os level rather than at the tool level.

Immediately before the actual `emacs-19.22` tool entry are some additional entries that make life easier. The simple entry `emacs` and the base version entry `emacs-19` are both entered as links to the highest stable installation version of `emacs v19`.

The standard entry is divided into three blocks. Each block is separated from the others by `| |`. The first two blocks are used by architecture masters. These masters are the hosts that are used to build and install the tool. The third block is used by clients.

The first block consists of entries to provide the tool to a host that can serve it from local disk. Since it comes from local disk, the type of entry is a link entry. The component `"host==terminus"` is a boolean test that is true only on the host `terminus`. The `"fs"` entry is the parent directory that the link should point to. The actual directory that is pointed to for `/tools/emacs-19.22` on `terminus` is `/disk/sd1f/tools/sun4-sos4/emacs-19.22`. Note that `$arch` and `$os` are replaced by the machine architecture and operating system identifiers that are built into `amd`.

The second block is used to provide the tool to an architecture master that does not store the tool on local disk. The mount options in the second block provide read write access to the mount point. Without write access, performing an installation would be difficult.

The third block provides client access to the installed tools. The NFS mounts are performed with the default options specified by the `/defaults` entry. Note that there are two entries that match if the host is a `sun4`. This provides a replication server since `AMD` will attempt to mount the corresponding directories at the same time. The machine that answers first will be used. If that machine should crash, `amd` will attempt to mount `/tools/emacs-19.22` from the alternate server.



# SENDS: a Tool for Managing Domain Naming and Electronic Mail in a Large Organization

*Jerry Scharf* – Sony Electronics  
*Paul Vixie* – Vixie Enterprises

## ABSTRACT

Managing the Domain Naming Service and electronic mail routing in a large organization has always been a difficult problem. Systems designed to automate these tasks encounter the basic difficulty that the information needed to maintain an accurate picture of the organization is distributed far more widely than the expertise needed to operate such a system.

This paper describes a simple set of tools that provides automatic central management of host and electronic mail information. It attempts to find a balance between centralized management and distributed autonomy by centralizing the tools and accumulated data and distributing the source of the data. It also centralizes the mail delivery technology. With this, the users and local groups are provided a higher level of service without the loss of control and local administration.

## Background

Sony Corporation of America's TCP/IP network evolved the way many other organizations did. Many groups purchased islands of workstations to meet specific needs. At first, the central networking organization ignored these machines as not central to the business, and everyone was happy. The use of Domain Name Service was spotty in deployment, integration, accuracy and quality.

Groups then discovered that they needed to share information, and the corporate networking organization was asked to make the systems work together. After a short period of time, the management of routers and subnets was well controlled, but the naming and electronic mail systems were still variable in coverage and quality. There was some amount of friction between the networking group and some of the more independent groups over control and ownership of information. It was also discovered that very few groups had any technical knowledge of DNS. Concurrently, Sony was starting to migrate applications off the IBM mainframes onto client/server platforms. These are Unix platforms communicating with TCP/IP. This has forced a whole new group of people to deal with the administration of machines and routed networking.

At the same time that this effort started, the central networks group needed to develop a common structure for the name space for the US network. When we discussed this with some of the key groups, all the discussions centered around the inextricable link of domain names and electronic mail addresses. At the same time, certain issues about the e-mail delivery systems came up repeatedly. The

most important issue was that it was very difficult to find the e-mail address of a given user. We realized that the management of e-mail should be folded into the DNS management system, providing the users with the increased functions that made the changes worthwhile.

## Problems to Solve

### Domain Naming Service Administration Issues

It is beyond the scope of this paper to describe DNS, but a brief introduction is needed to highlight some problems to be solved. DNS [1,6,7] was designed to be a hierarchical naming system to allow the ARPAnet to introduce tiered management of hostname to IP address translation. The tiers are produced by the "delegation" of a section of the naming tree to a subordinate administration and nameserver.

The delegation of administration by naming hierarchy worked well in the basic designs envisioned when the system was designed, but has since become problematic. Most notably the IP address to name lookup is based on the lexical structure of the dotted quad address, e.g. 10.0.0.1. With the significant use of subnetted networks and the development of Classless Inter Domain Routing (CIDR) [4] this mapping of administration to name often fails to align with the administrative boundaries of the organization.

At the administration level, electronic mail delivery is often considered a separate issue from providing DNS services. When the electronic mail is SMTP based, this is no longer true. DNS provides the mapping of addresses to hosts by the use of Mail

eXchanger (MX) records and Address (A) records. There is a further link in that it is often desirable for the mail delivery system to modify addresses in the mail header based on the address.

### Mail Delivery Issues

Every organization has a slightly different opinion of how SMTP [2,3,8] based mail delivery should be handled. In particular, the processing of headers is highly variable. At Sony, we added two features to the basic processing of mail headers.

The first feature was the ability to selectively remove the "host" from the "domain" in the mail headers. The reason for this to be selective is that it is often the case that groups have a common "aliases" file, in which case the host can be safely stripped. In the case where the local machine keeps a separate aliases file it is unsafe to strip the host, because `user@grouphost.domain` may map differently than `user@host.domain`, and replies to the message could go to the wrong address. We added a flag to the hosts file to indicate this condition, which is converted into a mapping database. It should be noted name mappings that occur in the user's mail reader are not a problem, since those are applied as the message is being built for delivery.

The second feature was the desire to transform the username on the left side of the @ to a `firstname_lastname`. This needed to be implemented on a group by group basis as groups often have duplicated usernames.

A final necessity is the ability to manage and forward deprecated addresses. One centrally maintained file keeps all this information, which facilitates maintenance, especially group name changes.

### Sony's Solution

The design of any system must reflect the basic structure of the organization. In Sony's case, there has been a strong push towards decentralization of control of the MIS functions as it related to systems and applications. These groups can range in size from five people to hundreds. The emerging client/server applications are moving away from the centralization of the mainframe. The networks, on the other hand, are centrally planned and managed. This means that any tool must account for the needs of the groups and network managers alike.

The lack of technical expertise in the individual groups was a key concern that led us to the set of tools we have now. In our survey of the other tools available for managing DNS naming, we found that they implicitly linked the loading of the data into the tool and the operation of the name server. Our goal became to find a way to allow the central networks group to operate a cluster of name servers based on information fed in by the distributed, autonomous groups. These groups, with their individual administrators, are the clients of SENDS.

At the same time, the staff of the Corporate Networks group was extremely overcommitted with

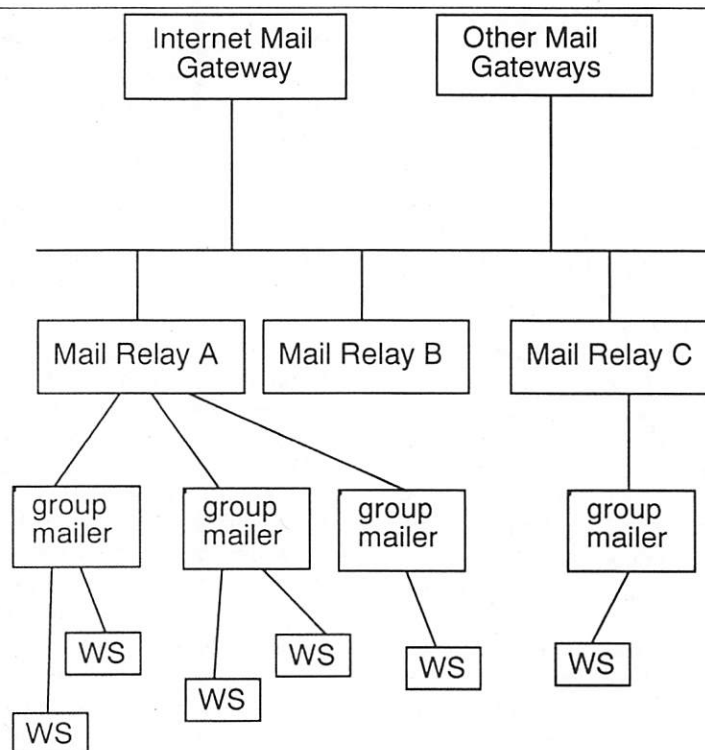


Figure 1: Mail Routing Diagram

the installation of routers and circuits that were being requested. There were no clerical resources to set up a call in line to get IP addresses registered. The groups were concerned that the speed of a clerical solution would hinder their local administration. We needed a tool that automated all the common actions in both normal and abnormal situations.

In looking at the problem of host registration, we realized that the same lack of expertise existed for the SMTP mail delivery. Some groups had very complex mail systems, some of which broke regularly, while others had no mail outside their group at all. Once again we needed to coalesce the mail delivery system into a task that a small number of people could maintain. The goal was to make minimal changes to the mail configuration on a group's mail server to incorporate it into the larger mail delivery system.

We settled on the idea of customizing a small number of machines and distributing them throughout the country as mail relays. These machines are owned and operated by Corporate Networks. The mail to and from the various groups flows through these machines. Each machine is identical in function and acts as a backup for other mail relays. They also provide DNS and NTP [5] time services to their area. We currently have 4 systems, each with the same software and databases, utilizing BSDI/Intel and HP platforms. See Figure 1.

Another direct benefit of centralized mail management is the ability to implement directory services. The form and content of information to be offered in a directory service will change over time, so the input format for the information must be extremely flexible. This is in sharp contrast to the exact information to be loaded into DNS. This led us to have two files, one for host information and the other for mail and directory information.

The lack of global directory services has always been one of the detractions of SMTP mail when compared to integrated mail systems, especially on the PC. Our directory service is bound to the mail delivery mechanism, which gives the groups a strong desire to keep it accurate.

To make the system as flexible as possible, the mechanism for processing the files must be separated from the structure that the namespace takes. The scripts are more complicated because of the fact that namespace overlap is allowed.

For each group that is managed under SENDS, the domains for the mail for the hosts are specified or inferred. The tools then generate all the DNS zone files and email mapping files based on these values. This allows groups to share host domains or email domains with other groups, while preventing naming conflicts. We have used this to allow the different corporate entities within Sony to structure their mail names to reflect their business structure.

In the last 10 years, many people put GUIs on management tools under the flag of ease of use. In our environment, with users spread from IBM mainframes to all flavor of Unix boxes to PCs and Macs, there is no common form of GUI. The only thing that can cover this range of platforms is a text based tool with character conversion. FTP does the conversion, so we use it as the transport with simple text files as the input format.

It also allows local groups to develop tools specific to their needs. They can generate the files with any editor on their local system. We also have groups that generate their files automatically from other data within the group.

### SENDS Processing

The heart of the system is the processing of the input files from the users. The tools are a group of *Perl* scripts run from cron. These scripts process the two files that each group maintains, the *hosts* and *users* files. The results of the tools are zone files for the primary DNS server, boot files for the primary and secondary DNS server, and database files distributed to the mail relay machines.

Each group administrator is given an FTP [9] only account on the registry machine. When the group administrator wishes to change the hosts or users files, they use FTP to drop a new version of the file into their FTP drop directory. SENDS processes each new file sequentially, sending mail with the results back to the group. We are using DECWRL's FTP daemon, and have it set up to chroot the admin directly to the ftp drop directory for the group. There is a restricted shell that only allows the group administrator to change the password of the ftp account.

The *hosts* file is the standard */etc/hosts* file used on Unix file systems. The *users* file is a freeform file with key:token pairs on separate lines which combine to make up records. The records are separated by blank lines, similar to uucp entries. A complete set of example input files and generated files are included in Appendix B.

### Directory Structure and Control Files

The top level of the SENDS directory contains the directories *data*, *bin* and *lib*. The *bin* and *lib* directories are for the tools, *data* is where the action is; see Figure 2. Under the *data* directory are two fixed directories, *Zones* and *Users*. *Zones* is where the final DNS information lives, and *Users* is where the final mail processing data resides. All subdirectories of *data* that start with a lower case character are treated as domains, and are assumed to have group files. All other directories are ignored.

The domains contain zero or more group directories and only those that start with lower case characters are searched. This allows the SENDS administrator to create a new group, beginning with a

starting upper case character so that it will not be processed while being set up. Within each group are the *control* file and four directories: *ftp*, *work*, *output* and *bad*.

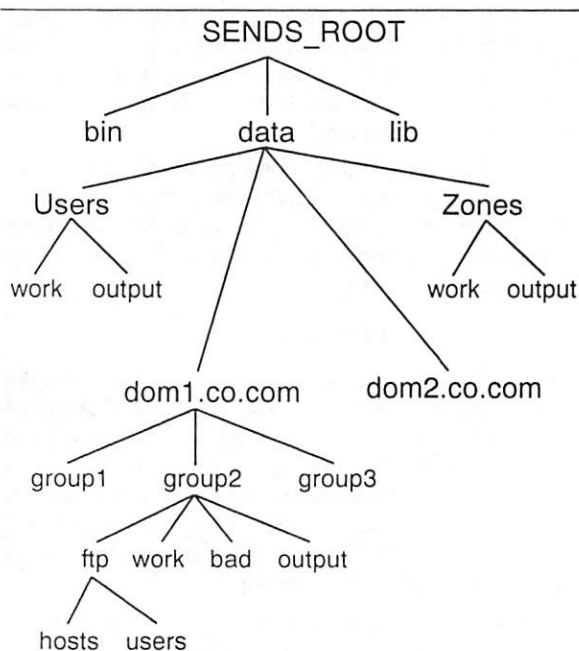


Figure 2: Directory Structure for SENDS

The common control features of the *Perl* scripts are put in a global file, *sends.conf*. The base directory for the SENDS tree is controlled by an environment variable, because it is needed before the config file can be located. The majority of the entries in the config file are template information used in the files generated by SENDS. It also includes the user record parsing requirements. The sample file is included in Appendix A.

The more important parameters are in the per group control file. This controls how the group's data is merged into the final output. The file is named *control*, and lives in the top level of the group directory tree. It has 6 keywords supported as described below.

The *network* lines are the most important in processing the group's host file. They take the form of Access Control List commands *accept* and *deny*, and there is an implicit deny all at the end of the list. We have adopted an unusual form to specify ranges, allowing the form (a-b) in each of the dotted quad locations. This allows ranges to be specified on any boundary, providing more flexibility than bit-masks. It allows the SENDS administrator to give a group a block of subnets with identical ranges inside each subnet. If, for example, addresses above 240 are reserved for routers, and the users aren't allowed to set the network name, this could be described by the one ACL line:

```
network: 10.0.(10-19).(1-239) accept
```

The *dri* is the mail address of a designated responsible individual. There can be multiple *dri* lines, and the results of processing any new group files are sent to all *dri*s as well as the SENDS notify address specified in the *sends.conf* file.

The *relay* keyword is used to specify mail relays for the group and backup mailers for all the hosts in the group. The value must be a fully qualified hostname including a trailing dot. The order of these statements is important, in that the first *relay* line gets MX preference 1, the second 2, and on. There can be more than one host on a *relay* line, in which case all the hosts on that line get MX records with the same value, allowing load sharing.

The *fqdn* specifies the place in the naming tree that the host entries will be placed, ie. *host.gr.mycorp.com* has the *fqdn* *gr.mycorp.com*. If this is not specified, it is generated from the directory structure below the data directory for SENDS, as *group.domain*. The *maildom* specifies the location of the group MX, the base address of the users in the users file, and the address the hosts are mapped to if they do not have the unique-alias flag set. The MX records for the host still go in the *fqdn* location. If a *maildom* is not set, it is assumed to be the *fqdn*.

The *dots-allowed* keyword accepts yes or no, and determines if hostname entries in the host file are allowed to have dots. Hostnames that tail match the current domain with just the host remaining are not considered to have dots. This allows the SENDS administrator to control whether groups are allowed to introduce new levels of hierarchy.

### Host File Processing

As was mentioned above, the hosts file is in standard Unix format. We have modified it slightly by permitting magic cookie comment that allows us to have flags for a host. Most important of these is the unique-alias flag, which prevents hostname to groupname mapping on the specific host. We have tightened some of the rules that we felt were ambiguous when converting to DNS. One goal was to allow users to use their existing host files as is, so the network ACLs in the control file are essential.

When a new host file is processed, we first process the hosts files of all the other groups, and construct a series of tables with names and IP addresses used. Then we read the current file. This enforces the first come, first serve allocation of IP addresses and names. Any syntactic errors in the file cause the update to be aborted, and the input file is moved to the bad directory. Other errors include: duplicate IP address, aliases (fully qualified) that match a basename or aliases of another host and a hostname or alias that is qualified but does not tail match the *fqdn* being processed. The hostname is the first name in the list after the IP address, and all the other names are aliases.

The IP address is checked against the network ACLs in the control file. If they do not end with an accept, a warning is produced, the line is ignored, and the processing of the file continues. This allows the group admins to have local additions to their hostfile without causing problems for the DNS namespace. Since almost every Unix hostfile has a 127.0.0.1 address in it, it would be a race to see which domain gets the reverse pointer, or every admin would have to have a separate file with these lines stripped.

Once the processing of a new file has completed successfully, a complete set of forward and reverse zone files for all groups is written into the work subdirectory of *data/Zones*, each with the SOA record removed. Each one is compared to the existing zone file, and if they have changes, the file is moved over and a new SOA file is created. We use the serial number format *yyyymmddnn* for all our SOAs. A new pair of boot files are produced, one for the primary server and one for a secondary. These are set up to be included from the *named.boot* file for the nameservers.

Finally for all the hosts that did not have the unique-alias flag, a file is produced that has the fully qualified hostname and the groupname to which it is to be translated. This is especially important if you have set up a namespace where hosts in one domain may be split among several mail domains.

#### Mail File Processing

Mail delivery for a group is controlled by the users file in the group. As mentioned above, we use a uumap style, in which a record is built up of key: value pairs, one per line, and a blank line separates records. This is a very general format that allows us to have many types of optional data and allows for future format changes.

There are currently three required keywords for each user: *username*, *mailname* and *mailhome*. The *username* is the name of the account to which mail is sent and received within the group, and it must be unique within the group. The *mailname* is the name to which the *username* is transformed when it passes through the mail relay, and also must be unique within the maildom. It is fine to have the *username* and *mailname* be the same, but both must be specified. The *mailhome* is the host to which mail for this user is to be delivered, and must be a fully qualified hostname. This allows different machines to receive mail for subsets of the users within a group.

A special record is supported with the *mailname* and *username* both set to *""*. This is the place where all mail to a group that fails to match either a *username* or a *mailname* is sent. This allows mail aliases to work without explicit listing in the users file.

In addition to user records we support a record with a *default* keyword. The idea is that the default record is appended to each record that follows, with any duplicate keys being set to the user record's value. This allows much of the non-unique data in the file to be specified only once. There are two values for the default key, *replace* and *merge*. *Replace* clears the current default record and adds the contents of this record. *Merge* adds or replaces the keywords in the current default record.

Here is an input users file:

```
default:      replace
organization: usenix
scope:        US
mailhome:     mail1.us.lisa.usenix.org

username:     scharf
mailname:     Jerry_Scharf

username:     vixie
mailname:     Paul_Vixie
mailhome:     wsl.us.lisa.usenix.org

default:      merge
scope:        everywhere

username:     nile
mailname:     Martin_Nile

default:      replace
mailhome:     prim.us.lisa.usenix.org

username:     brister
mailname:     James_Brister
```

and here is the expanded output after processing:

```
mailname: Jerry_Scharf
username: scharf
mailhome: mail1.us.lisa.usenix.org
scope: US
organization: usenix

mailname: Paul_Vixie
mailhome: wsl.us.lisa.usenix.org
username: vixie
scope: US
organization: usenix

mailname: Martin_Nile
username: nile
mailhome: mail1.us.lisa.usenix.org
scope: everywhere
organization: usenix

mailname: James_Brister
username: brister
mailhome: prim.us.lisa.usenix.org
```

It is important to notice that all groups that have the same maildom must be processed at once and all uniqueness tests are done across a mail domain, not a single group.

Once all the files for all the mail domains are loaded, a file is generated for each maildom with each user having a fully expanded record. A three columned file is produced specific to the mapping and sendmail configuration we support at Sony. The first column is `username<@groupaddr>`, the second `mailname<@groupaddr>` and the third one `username<@mailhome>`. The mapping from column one to column two is used on mail leaving the group, and the mapping from column two to column three is for mail going to the group. The reason for the difference between column one and column three is that we want to remap the username from any machine in the group, so we first map the `user@host.group.dom` to `user@group.dom`, then map that to `mailname@group.dom`.

### The Mail Delivery System

For each user in each group, a mail target machine is defined. The mail targets for the groups are a list of mail relays. Mail destined for a specific machine is delivered directly to the machine. Mail destined for the group is delivered to a mail relay, which looks up the address, determines the machine and account name it should be sent to, and sends it on.

The current system uses IDA sendmail with *dbm* support and custom *sendmail.cf* files to do the work. The *dbm* databases provide the following mappings:

```
hostname->groupname
    for header rewriting
user@groupname->fullname@groupname
    for header rewriting
user@groupname->user@mailhost.groupname
    for envelope rewriting
fullname@groupname->user@mailhost.groupname
    for envelope rewriting
obsolete_address->new_address
    for header and envelope rewriting
```

### Directory Services

Our first directory service is a *whois* server based on a simple *Perl* script. Each night, the expanded users database is processed, merged with data from other mailers that have SMTP attachment, and written to a flat file with artificial handles. The *Perl whois* server does something similar to a *grep* of the file for the string, and returns matches.

### Deployment

SENDS was put into production October of 1993. The current revision has been running since April of 1994. We have in that time been supporting two separate naming structures simultaneously.

SENDS is managing about 50 distinct groups inside Sony in the US. There are 3 separate corporate entities using it, each with their own

substructure. There are approximately 250 DNS zones generated, 500 users and the obsolete address forwarding list is about 1300 entries. These all seem modest from our past experience, and we have had no serious scaling problems to this point.

The regional mail relay systems are too lightly loaded to show any penalties of the database lookups, passing about 2000 messages a day. We will need to get in the range of 2000 messages per hour per machine before we can study those effects.

We have set up the cron job to run the file scanner every 10 minutes. Our DNS records have a Time To Live set at 30 minutes for both zones and cache entries. This guarantees convergence in 70 minutes from when a file is submitted, with the average in the 20-30 minute range. Our mail databases are loaded hourly, so we have a similar convergence time there.

Our experience in setting up novice groups to run SENDS and mail have been very good. If we have seen the O/S and version before, it is usually one to two hours to create the groups in SENDS, load the hosts and users files, set up one group mail server for sending and receiving and let the data propagate through the network.

### Conclusions

SENDS is a simple and powerful tool for maintaining the DNS namespace and e-mail administration. SENDS has proven ample for Sony's current and near term needs. The groups are pleased with the response time to changes, and Corporate Networks staff are pleased with a reduction of workload in managing and tracking of DNS names. The integration and ease of use has also caused the groups to be more diligent at maintaining their data.

We would recommend SENDS to any organization who had more than a few separate administrative groups with IP machines. The key assumption of this design is that the individual groups maintain control of allocation of their section of the IP and DNS spaces. If that doesn't follow the model your organization wants, SENDS is not the tool for you.

The directory services are still new at Sony and not fully developed. We will need to develop a better idea of which information is needed in the directory, and which access methods work best. There will need to be a tool that will allow the group systems administrator to delegate the directory maintenance to the individual users. We have tried to make the system independent of naming structure, but this will only be tested when others attempt to implement their designs with these tools.

We have been pleasantly surprised a couple times by what we have been able to persuade SENDS to do. Though we would not recommend using SENDS as a mailing list administration tool, we set up a pager maildom with a list of users and their

Skyword pins, so people can mail to scharf@pager... and get a message to me.

Because Sony's use of TCP/IP based machines is modest, we have kept the tools simple at the cost of speed. We can process all of our data in under 1 minute, so things like caching *Perl* internal tables have not been done. If someone needs to manage a set of machines many times the size, this should be considered.

#### Acknowledgements

We would like to thank several people. First and foremost we would like to thank Sandi Resnik-off, whose skills and commitment to quality have allowed us to accomplish what we have. James Brister did major work on the code. Tim Guarnieri, Alison Nicklin and Cindy Goral did proofreading, blame us for the mistakes, thank them for the quality. Thanks also to Martin Nile, PJ DeFrance and the San Jose networks group for making this all work.

#### Availability

SENDS is available with a license that says that changes must be provided back to the maintainers and may not be resold. This is to keep the number of forks to a minimum. The files are available for anonymous FTP from ftp.vix.com in /pub/vixie/SENDS/sends.tar.gz

#### Author Information

Jerry Scharf is currently Strategic Planner for Corporate Networks at Sony Electronics Corp, and also manages Sony's Internet gateway. He has also worked as a system administrator and planner at Digital Equipment Corp., Nasa Ames Research Center and Cornell University. He did his undergraduate studies in physics at Pennsylvania State University. Reach him via US Mail at 3300 Zanker Rd, San Jose, CA 95134. His electronic mail address is <scharf@sony.com>

Paul A. Vixie is a system, software and network consultant in the San Francisco Bay Area. From 1988 to 1993 he worked for Digital Equipment Corporation's Western Research Laboratory ("DECWRL") where among other things he catapulted the largely unknowing and unwilling corporation into the Internet limelight by creating gatekeeper.dec.com, now one of the largest and busiest Anonymous FTP hosts in the world. He is the author of BSD's cron daemon and of King James Sendmail; he is also the current maintainer of the BIND domain name server. Reach him via US Mail at 116 Stanley St., Redwood City, CA 94062, USA. His electronic mail address is <paul@vix.com>

#### References

- [1] Paul Albitz and Cricket Liu, "DNS and BIND," O'Reilly & Assoc.
- [2] Bryan Costales, Eric Allman and Niel Rickert, "Sendmail," O'Reilly & Assoc.
- [3] D. Crocker, "RFC822: Standard for the format of ARPA Internet text messages,".
- [4] V. Fuller, T. Li, J. Yu, K. Varadhan, "RFC1519: Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy".
- [5] D. Mills, "RFC1305: Network Time Protocol (v3)".
- [6] P. Mockapetris, "RFC1101: DNS encoding of network names and other types".
- [7] P. Mockapetris, "RFC1035: Domain names - implementation and specification".
- [8] J. Postel, "RFC821: Simple Mail Transfer Protocol".
- [9] J. Postel and J. Reynolds, "RFC959: File Transfer Protocol".

## Appendix A: SENDS config files

An example global config file:

```
# Config file for SENDS.
#
# address of primary nameserver for the zones we control. A dotted-quad is
# REQUIRED
primary          10.0.1.10
zonemask          10.0.0.0 0xffffffff00
zonesize          10.0.0.0 3          # in octets
# fields in a user entry that may not have default values REQUIRED
user_key_no_defaults  username mailname
# fields in a user entry that may not have duplicates REQUIRED
user_key_no_dups      username mailname
# fields in a user entry that have to exist REQUIRED
user_key_min_fields   username mailname mailhome
# names of the name servers (primary and secondary) for our zone.
# These entries go in the zone boot file for named.
name_servers          prim.us.lisa_test.usenix.org. sec.us.lisa_test.usenix.org. sec.t
hem.lisa_test.usenix.org.
# hostmaster entry for the SOA record. REQUIRED
hostmaster            hostmaster.lisa_test.usenix.org.
# master host for the zones SOA record. REQUIRED.
zone_master           prim.us.lisa_test.usenix.org.
# zone expiration value. OPTIONAL--default is 604800
zone_expire           604800
# zone retry value. OPTIONAL--default is 600
zone_retry            600
# zone refresh value for the SOA record. OPTIONAL--default is 1800
zone_refresh          1800
# zone minimum value for SOA record. OPTIONAL--default is 1800
zone_minimum          1800
# directory for putting temp files. OPTIONAL--/var/tmp/is default.
#tmpdir               /var/tmp
# address for mailing copies of processing log and data file
# notifications. REQUIRED
notify_addr           scharf
# location of sendmail binary. OPTIONAL--default is first value
# found on PATH.
sendmail              /usr/sbin/sendmail
# predefined domains that hostnames cannot match against. OPTIONAL
bad_domains           foo.gov
# debug               1
#dont_mail            1
whois_file            data/whois
```

An example group control file:

```
dri:                  scharf@us.lisa.usenix.org
dri:                  vixie@us.lisa.usenix.org
relay:                mail1.us.lisa.usenix.org.
relay:                mailx.them.lisa.usenix.org.
```

```
network:      10.0.(0-30).(0-254)    accept
dots-allowed: no
```

### Appendix B

These are sample input and output files for the group "us" in domain "lisa.usenix.org". This was run with the control files in Appendix A.

Host file for the group "us":

```
127.0.0.1      localhost loghost
#
10.0.1.0       service-net      # this is a comment
10.0.1.10      prim dns_root ns
10.0.1.11      sec dns_sec
10.0.1.12      maill
10.0.1.20      big_serv nfsserv nfsserv_ef0
#
10.0.10.20     big_serv nfsserv nfsserv_ef1
10.0.10.30     wsl
```

The users file for the group "us", and was used as the example of the default record expansion, and can be found in the Mail File Processing section of the paper.

This is the mail file sent for processing the hosts file in "us":

```
To: scharf@us.lisa.usenix.org, vixie@us.lisa.usenix.org
Cc: scharf@morality.sjc.hw.Sony.COM
Subject: new "hosts" file for [lisa.usenix.org, us] received by SENDS
```

```
----- installation
/home/scharf/net/LISA/testrun/data/lisa.usenix.org/us/work/hosts:1:
  127.0.0.1 is not in your network ACL (ignored)
  127.0.0.1 localhost loghost
----- end of installation
succeeded - accepting input file
```

Here is the DNS zone file for us.lisa.usenix.org:

```
; this file was automatically generated - do not edit
$ORIGIN us.lisa.usenix.org.
@                IN MX    1 maill.us.lisa.usenix.org.
                 IN MX    2 mailx.them.lisa.usenix.org.
localhost        IN A      127.0.0.1
service-net      IN PTR    0.1.0.10.in-addr.arpa.
prim             IN A      10.0.1.10
                 IN MX     0 prim
                 IN MX     1 maill.us.lisa.usenix.org.
                 IN MX     2 mailx.them.lisa.usenix.org.
dns_root         IN CNAME  prim
ns               IN CNAME  prim
sec              IN A      10.0.1.11
                 IN MX     0 sec
                 IN MX     1 maill.us.lisa.usenix.org.
                 IN MX     2 mailx.them.lisa.usenix.org.
dns_sec          IN CNAME  sec
maill            IN A      10.0.1.12
                 IN MX     0 maill
                 IN MX     1 maill.us.lisa.usenix.org.
                 IN MX     2 mailx.them.lisa.usenix.org.
big_serv         IN A      10.0.1.20
                 IN MX     0 big_serv
                 IN MX     1 maill.us.lisa.usenix.org.
                 IN MX     2 mailx.them.lisa.usenix.org.
```

```

nfsserv          IN A      10.0.1.20
                  IN MX      0 nfsserv
                  IN MX      1 mail1.us.lisa.usenix.org.
                  IN MX      2 mailx.them.lisa.usenix.org.
nfsserv_ef0      IN A      10.0.1.20
                  IN MX      0 nfsserv_ef0
                  IN MX      1 mail1.us.lisa.usenix.org.
                  IN MX      2 mailx.them.lisa.usenix.org.
big_serv         IN A      10.0.10.20
nfsserv          IN A      10.0.10.20
nfsserv_ef1      IN A      10.0.10.20
                  IN MX      0 nfsserv_ef1
                  IN MX      1 mail1.us.lisa.usenix.org.
                  IN MX      2 mailx.them.lisa.usenix.org.
ws1              IN A      10.0.10.30
                  IN MX      0 ws1
                  IN MX      1 mail1.us.lisa.usenix.org.
                  IN MX      2 mailx.them.lisa.usenix.org.

```

Here is the header file for the zone us.lisa.usenix.org:

```

; this file was automatically generated - do not edit
$ORIGIN us.lisa.usenix.org.
@      IN SOA   prim.us.lisa_test.usenix.org. hostmaster.lisa_test.usenix.org. (
                                1994072900      ; Serial
                                1800             ; refresh
                                600              ; retry
                                604800           ; expire
                                1800             ; minimum
                                )
      IN NS     prim.us.lisa_test.usenix.org.
      IN NS     sec.us.lisa_test.usenix.org.
      IN NS     sec.them.lisa_test.usenix.org.

```

\$INCLUDE sends/us.lisa.usenix.org.data

This is the reverse lookup zone for subnet 10.0.1.x:

```

; this file was automatically generated - do not edit
$ORIGIN 1.0.10.in-addr.arpa.
0.1.0.10.in-addr.arpa.  IN PTR   service-net.us.lisa.usenix.org.
                        IN A      255.255.255.0
10.1.0.10.in-addr.arpa. IN PTR   prim.us.lisa.usenix.org.
11.1.0.10.in-addr.arpa. IN PTR   sec.us.lisa.usenix.org.
12.1.0.10.in-addr.arpa. IN PTR   mail1.us.lisa.usenix.org.
20.1.0.10.in-addr.arpa. IN PTR   big_serv.us.lisa.usenix.org.
                        IN PTR   nfsserv.us.lisa.usenix.org.
                        IN PTR   nfsserv_ef0.us.lisa.usenix.org.

```

Here are the primary and secondary boot files to be included from named.boot.

```

; this file was automatically generated - do not edit
; it is intended to be included from a named.boot file
primary 1.0.10.in-addr.arpa      sends/10.0.1
primary 10.0.10.in-addr.arpa     sends/10.0.10
primary us.lisa.usenix.org       sends/us.lisa.usenix.org

; this file was automatically generated - do not edit
; it is intended to be included from a named.boot file
secondary 1.0.10.in-addr.arpa    10.0.1.10 sends/10.0.1
secondary 10.0.10.in-addr.arpa  10.0.1.10 sends/10.0.10
secondary us.lisa.usenix.org     10.0.1.10 sends/us.lisa.usenix.org

```

Finally, here is the three columned file used for mail address transformations. When mail is sent out from a group, sending addresses that match column 1 are replaced with column 2. Inbound mail that matches either column 1 or column 2 are sent to the address in column 3.

```
scharf<@us.lisa.usenix.org> Jerry_Scharf<@us.lisa.usenix.org> scharf<@maill.us.lisa.usenix.org>  
vixie<@us.lisa.usenix.org> Paul_Vixie<@us.lisa.usenix.org> vixie<@ws1.us.lisa.usenix.org>  
nile<@us.lisa.usenix.org> Martin_Nile<@us.lisa.usenix.org> nile<@maill.us.lisa.usenix.org>  
brister<@us.lisa.usenix.org> James_Brister<@us.lisa.usenix.org> brister<@prim.us.lisa.usenix.org>
```



# Getting More Work Out Of Work Tracking Systems

*Elizabeth D. Zwicky – Silicon Graphics*

## ABSTRACT

This paper discusses work initially done for SRI International's Information, Telecommunication, and Automation Division (ITAD). ITAD's computer facility staff originally implemented a work tracking system to avoid the embarrassment of discovering that some important user problem had been brought to their attention and then entirely forgotten. Over the years, the system also began to address more complex tasks, and is now used to deal with some problems before users report them, to better communicate with the users about the amount of work being done, and to get those minor housekeeping chores that kept accumulating done at last. This paper explains how.

### Work Tracking Systems

Work tracking systems, also commonly known as trouble ticket systems, are systems that keep track of user problems for a group of people. These have a family resemblance to the systems used to track bugs in software (for instance, they generally allow you to assign responsibility to a person, set priorities, and open and close entries), but differ in some basic assumptions (for instance, bug tracking systems generally assume that a bug can be assigned to a particular product, that it only needs to be fixed once, and that the database of bugs will be searched by naive users, while work tracking systems generally assume that the problem may have multiple causes, that it may recur in other places, and that the database will be searched only by the people fixing things). There are both commercial and public domain systems available, at various levels of complexity and specialization for system administration. At the most complex end, problems are submitted through a special program with a graphical user interface and tracked in a relational database. At the simplest end, problems are submitted through electronic mail and tracked with some pseudo-database.

### ITAD's Tracking System

ITAD's tracking system, internally called the "action queue", is very much at the simpler (and cheaper) end of the scale. It was developed at ITAD but modeled on the University of Colorado's QueueMH. ITAD's system is discussed in [7] and Colorado's in [2]. Users submit problem reports in e-mail, and the people dealing with them use modified MH commands to set priorities, list outstanding problems, update reports, and close them when the problems have been fixed. This is well suited for ITAD's environment, for several reasons: ITAD considers it important for users to be able to submit problems directly into the work tracking system; MH is already the standard mail system; and

there is a need to support users on a wide range of platforms, making it difficult to port beautiful graphical user interfaces. Some of ITAD's users are also strongly resistant to learning new programs, however beautiful, and since they all already knew how to use some mail system, and were accustomed to sending electronic mail requests to "action", maintaining this interface kept them happy.

### Why Get Fancy?

In late 1992, ITAD's computer facility staff became embroiled in yet another discussion of exactly what it is that the computer facility does and why it takes so long. With four full-time system administrators and only a few problems coming in per day, people didn't understand why problems weren't fixed immediately. The system administrators didn't understand why nobody had yet snapped under the strain, since in addition to the few hundred items that were in the queue, they were dealing with all the maintenance and upgrades that weren't direct user requests, plus all the telephone requests, questions in the hall, and notes taped to doors.

The computer facility developed a new purpose in life; putting absolutely everything that needed to be done into the action queue. When this initiative started, there were a few hundred items in the queue, and 10 to 20 of them were resolved each week. Currently, there are about 1,200 items in the queue, and approximately 150 are resolved each week. (At the queue's peak, there were more than 2,000 queue items.) Even the most skeptical of users is capable of figuring out that this means that the computer facility does a lot of stuff, and has even more stuff left to do.

The change took several months, and required new software and new procedures.

### The Software Side

Aside from the action queue software itself, there is a set of programs I call "complainers". Each one of these programs detects problems in some particular area; for instance, one of them compares the nameserver's hostname to address mappings with its address to hostname mappings to make certain they match. These programs output messages, separated by a blank line. A separate program, imaginatively named "complain", takes messages separated by blank lines as input, compares their subject lines to those of messages currently in the queue, and submits them if they are not already present. This makes certain that any given problem is reported only once. `complain` also introduces a 5 second delay between messages to avoid overrunning the mail queues. (This is a common problem with programs that are automated mail senders, since they tend to be lightweight compared to the mail system. A good, up-to-date `sendmail` configuration will keep you from drowning the receiving machine, but not from using up all the memory on the sending machine.)

These programs are run from `cron`, via a small script which has multiple names. It looks for a directory in `/usr/facility/complainers` that is named after the script, runs all the programs in it, and pipes the output to `complain`. It is run every evening as "complain-daily" and once a week as "complain-weekly".

Currently, the daily complainers are `bad-aliases`, `bad-time`, `dumps`, `today-errors` and `user-problems`. The weekly complainers are `hosts-not-hosts`, `missing-man-pages`, `multi-mount` and `no-modules`.

#### `bad-aliases`

`bad-aliases` runs through the `aliases` file, looking for aliases that include files that don't exist, refer to local users that don't exist, or forward mail to remote hosts that return "User unknown" for the user we're forwarding to. For local lists, it reads through the files; for remote hosts, it uses a local program called `mailaddr` that contacts the remote host and does an SMTP VRFY. It would be nice to weed out forwards to non-existent hosts, but there is no easy way to distinguish between a host that doesn't exist and a host that isn't reachable at the moment. `bad-aliases` also reads the comments in our files looking for a date, and complains if the date is more than 6 months old; we use this for mailing list aging.

#### `bad-time`

Theoretically, all of ITAD's machines run `ntp` or otherwise synchronize clocks. In practice, `ntpd` occasionally dies, and some users have root, modify `rc.local` in odd ways, and fail to notice that the time is wrong until they call up complaining that make is behaving oddly (because they're 8 minutes off the

time their file server has). `bad-time` checks that all hosts are within 5 minutes of the time on the host running the complainers. It ignores hosts in a different time zone from the host it's running on. `bad-time` is ITAD-specific only in the code that identifies the hosts to run on and the specific message it outputs; otherwise it only needs to be able to run `date` on the remote hosts. Any site that has a simple way to get all remote hosts to provide the time in number of seconds since the epoch, instead of in human-readable form, would probably be much better served to rewrite the program, since that would be simpler and more reliable than `bad-time`'s method of parsing the human-readable date. (It would be a service to the universe if somebody managed to make it standard for `date` to provide a format code for "I don't want a beautiful date, just give me the seconds". I have known sites that simply installed a `stupidddate` program for this purpose.)

#### `dumps`

`dumps` is another doublecheck. ITAD's backup system sends mail when it's working and when it encounters errors, but this complainer checks the dump frequency field against the data in `/etc/dumpdates` as a doublecheck. This will catch, among other things, disks that have been assigned to a dump run that is never performed. The basic concept behind `dumps` is applicable to any backup system that uses `dump` as its transport agent and runs on machines with a dump frequency field in `fstab`, but code changes will be necessary to adapt to the values other sites use in the `fstab` dump frequency field. It could be adapted to any situation where the information about what should have been backed up and what was backed up is available to a program other than the backup system.

#### `today-errors`

`today-errors` is one of the most complex and useful complainers. It reads through today's messages in the `syslog` or equivalent message log on each of ITAD's machines, sorting the messages into three classes; messages it knows are ignorable, messages it knows are bad, and messages it doesn't recognize. Ignorable messages include normal boot messages, informational messages from programs that insist on logging their startup or automated restart (`routed`, `mrouted`, and `named` for instance), and messages about missing services that occur while the servers are down for backup. Messages which are recognized as specific problems with specific advice provided include disk errors, `lpr` complaining about missing printers, and memory errors. In addition, it counts the number of reboots encountered, and sends a tailored message if it exceeds 3. Messages that are not recognized are sorted by the program that logged the message, and one action request is queued for each program.

This finds a number of problems that might not otherwise have been noticed, ranging from dying SCSI disks, which are now usually replaced **before** they completely fall apart and the user has no machine, to a user who believed that the right way to debug his X environment was to power-cycle his machine every time it hung. It also has discovered a number of things were perhaps better left concealed; both the `routed` and the `sendmail` version ITAD currently runs have bugs that cause them to log incorrect error messages, for instance. Then there are the mysteries of the universe, like the machine that logs "Sun 4/600" once every few months. That's the complete line. The machine is not a 4/600.

`today-errors` is highly ITAD-specific, in that not only the code for figuring out which machines to run on, but also all of the regular expressions and tricks it uses to classify lines, are hard-coded into the program.

#### **user-problems**

`user-problems` compares `/etc/passwd` to what it can find of reality. For instance, ITAD has a centrally maintained on-line phone database, which `user-problems` compares to the phone number and location shown in `/etc/passwd`. `user-problems` also checks to make certain that users with valid passwords have valid home directories, that their mail is forwarded somewhere, and that they have calendars; conversely, it attempts to assure that users with "\*" passwords do not have valid home directories, do not receive mail locally, and do not have calendars. (ITAD uses "\*" to indicate users whose accounts have been removed, and "\*\*\*\*" for administrative accounts that the program should ignore.)

This serves partly as a check to make certain that people have installed accounts using the programs for that purpose, instead of by hand, and partly as a check on removing accounts, which we prefer not to automate. While it mostly catches botched installs, it also regularly catches people who have changed offices or phone numbers, inadvertent corruption in the online database, and other forms of bit- or brain-rot.

`user-problems` is extremely ITAD-specific, since it knows a great deal of detail about where things are, how they are formatted, and what different kinds of users should have. For instance, it enforces a specific GECOS format, including an expiration date, and cross-checks that against a particular file parsed by column position; it also insists that all users be on one of two specific mailing lists. Other sites are unlikely to find it useful except conceptually.

#### **hosts-not-hosts**

`hosts-not-hosts` compares `/etc/hosts` to the nameservice databases, and also compares forward and reverse data within the nameserver. When it first ran, it found a surprising number of hosts

registered in only one place, or registered differently in A and PTR records. Once these were removed, people learned not to do that any more, and `hosts-not-hosts` rarely if ever finds any differences at this point, although there is still no automated program for adding hosts that would avoid these problems. It also compares the NIS ethers file to the hosts in the nameserver. ITAD considers the nameserver the primary source of host information, not the hosts file, and it maintains an ethers file for all hosts, regardless of whether they will ever need to use RARP. This is partly to aid in diskless booting of hosts that normally run dataless but don't at the time have functional operating systems locally, and partly to aid in the identification of hosts from packet traces on the network.

`hosts-not-hosts` is fairly general; it requires a hosts file containing only local hosts, tries to run `named.xfer` on the zone the host it's running on is in, and tries to `ypcat` the ethers map. If those conditions can be met, only a few strings need to be changed.

#### **missing-man-pages**

`missing-man-pages` compares programs in `/usr/local/bin` to manual pages in `/usr/local/man`, and complains when programs do not have matching manual pages. With programs like Perl and GNU Emacs, which want to install a version-numbered link, this requires you to link the manual pages as well; I haven't yet figured out whether I think this is a good thing or not. While it might possibly be helpful to users, there's no evidence that they ever notice, and it clutters `/usr/local/man` with links. The complainer has increased the likelihood that programs will have manual pages (although in practice not the likelihood that the program's installer will put them in place originally), but not to a certainty. Since it walks through all the binaries anyway, it also complains about compressed binaries (the complaint says neutrally that they are compressed and thus useless; some people choose to uncompress them, others to remove them). It should work at any site with minor and obvious changes to the particular binary and manual page directories.

#### **multi-mount**

ITAD's environment has an unusually large number of networks, and machines move from one to another with unusual frequency (particularly when you realize that machines are rarely physically moved). This sometimes results in machines receiving NFS service from multiple servers in situations where they could be using a single server. This cannot be entirely fixed by using automounts, since some of the file systems involved are not interchangeable, and furthermore many of the users complained that even though automount would normally give them the closest server, it was not guaranteed to be deterministic; they wanted the server closest in

network terms, not currently fastest in responding, and they wanted it to be the same every time. multi-mount detects cases in which the user's email, the user's home directory, and the /usr/local mounted on the user's home machine are not all three on the same NFS file server. This is unlikely to even be an interesting question at most sites, but except for the code that determines the host that holds the user's home directory, it's perfectly portable.

#### no-modules

no-modules detects software packages installed in their appropriate central location which do not also have files for use with the Modules package, and which do not have ".nomodule" files in their top directory. Modules, described in [1] allows users to add programs to their environment easily. With directory names changed, it should work for most sites using Modules.

#### Why Complainers and Not Fixers

People usually ask why ITAD uses complainers and not fixers. In fact, there are fixers for some problems, but many problems are not amenable to automated solutions. A number of the complainers are backups to automated systems that are supposed to have fixed or avoided the problem already, but which occasionally fail or are circumvented. Others involve situations where either of two information sources might be correct (for instance, the password file or the phone database might have the correct phone number) and there is no automated way to distinguish. Some of them clearly are very risky to attempt to fix automatically; for instance, an invalid password entry with an existing home directory should not automatically be fixed by removing the home directory. Occasionally there is a complainer instead of a fixer simply because the complainers are much faster and easier to write, and serve to keep the facility in line while people look for the time to figure out how to fix or avoid the problem.

Complainers also have more applicability for the environment at Silicon Graphics, where the hosts have considerably autonomy, and it's appropriate for system administrators to check for problems, but not for them to actually make changes to other people's systems.

#### ITAD's Experiences With Complainers

In general, the complainers are an improvement. Problems are detected earlier, more often fixed before users complain about them, and things are generally tidier and more comprehensible. On the other hand, maintaining the complainers can be difficult, and has required some changes to the facility solely to make the complainers work (for instance, the use of \* and \*\* passwords). Furthermore, it has proven to be difficult to get people to deal with complaints consistently and correctly. People have a tendency to remove items that they know

are not problematic each time they occur, rather than trying to make the complainers stop complaining. People often fail to check for earlier occurrences of errors in order to merge new complaints with them, possibly because it's an action that's extremely rarely required except for items from the complainers. Real people rarely send in new complaints about the same problem with different subject lines from the original (and when they do, the two complaints are almost never correctly merged together).

There is also an unfortunate tendency to treat the complainers as humans. I've seen many people remove a complaint with an explanation of why the complainer's interpretation of the problem is wrong. (For instance, a complaint about a missing manual page may be removed with the notation that the manual page exists under another name.) Oddly enough, the software does not generally listen to such explanations. The instructions that it provides are not always correct either, and people have been known to take phrases like "this probably means that ntp is not running" as gospel (as per the famous saying, "garbage in, gospel out"), leading them into trying to figure out what the ntp problem is when the real problem was failure in the complainer to compensate for times that cross an hour boundary, and the clocks were not off. This is despite the fact that the message clearly lists two times less than five minutes apart, with the notation underneath that they are more than five minutes apart.

#### Underpinnings of the Complainers

Many of the complainers rely on the ability to find all hosts of particular types, which ITAD provides by maintaining a file called Hosts.Status listing various information about our hosts including hardware type, OS type, and a classification into server, dataless client, diskless client, and standalone. This file is built automatically from the nameserver database, using HOST and TXT records to keep the information, and contains a field marking whether the host was "up" or "down" (in reality, whether it responded to ping) last time the file was built. Hosts.Status is built automatically every night, and is used by a number of programs besides the complainers; it actually predates the complainers by several years.

#### Software Other Than the Complainers

The complainers are not the only programs that insert items into the action queue. Any program that can send mail can insert items into the queue, and ITAD tries to configure things as much as possible so that anything that needs administrative assistance sends mail directly into the queue. In particular Wietse Venema's TCP wrappers [9] mail information about refused TCP/IP connections directly into the queue. This simplifies detection of break-in attempts, at the cost of severely reducing our patience with people who insist on trying failed

connections multiple times. Other security programs also send mail directly into the action queue; this allows us to deal with problems which should be looked at, but would be far too frequent and annoying to page a specific security person about.

#### Changes to the Underlying Trouble-Ticket System

The trouble-ticket system that was perfectly adequate with 200 items in the queue was not perfectly adequate with 2,000 queue items. We ended up making several after-market modifications. The first was to dramatically enhance the available reporting options, so that as well as getting a list sorted by submission date of the items assigned to you, you could get a list sorted by priority, and within priority have user-submitted tickets before auto-generated tickets, and within that have them ordered by the time since they were last responded to. This corresponds roughly to order of importance. Another report lists goals and the extent to which they were being met. For instance, one goal was to have each ticket assigned to a group and given a priority; another goal was to have no tickets over a year old.

There is also a staggeringly unpopular program called *whine* that sends you electronic mail telling you how many of your action items haven't been replied to recently enough for their priority level. This mechanism works beautifully for people who occasionally have a few overtime items. If you have people who routinely have several hundred, forget it; anyone with any sense simply puts mail filtering in. People who occasionally have several hundred are responsible for the bulk reply and bulk remove features that were added to the trouble ticket system (and are supposed to be used only on autogenerated items, as it's hard to come up with a bulk reply that works reasonably for human-generated items.)

#### The Human Side

ITAD made two fundamental changes in its procedures. First, everybody became fanatical about making certain that all requests from users went into the queue, no matter how the user submitted the request. Doing this is very difficult; it's hard to remember, it increases the time that short requests take, and it requires that you figure out how to restate the user's question so that it's immediately comprehensible, easy to type, and not insulting to the user. (It also required yet another change to the trouble ticket system to allow tickets to be submitted in a closed state.) On the other hand, it makes all those requests visible. Those little questions can eat away a day, leaving you feeling like you got nothing done whatsoever; the documentation proves that something did happen. When one of the ITAD users complained that the computer facility wasn't responsive enough to his requests, the queue logs proved he submitted more requests than anybody else in the division - as many, in fact, as the next several

people added together - and this made management considerably less receptive to his complaints.

Second, everybody entered the things that ought to be done, even though users hadn't requested them. This exercise was enlightening, if not in the end terribly encouraging. Taking the time to consider exactly what needed to be done, independent of current crises, was an enjoyable and fruitful exercise. On the other hand, it eventually became clear that the volume of crises was such that a lot of these projects were never going to get done, no matter how useful they would be. This was not only inherently depressing, but also a major disincentive to put the queue items in, since they merely sat around for a year or so before getting removed. The final compromise position is that the facility staff now enters into the queue only those projects that they consider absolutely critical, not the ones that they consider merely highly desirable. This makes only a moderate difference in the number of them that actually get achieved, but a significant difference in the usability of the statistics in staffing arguments.

#### Results

This enterprise did have most of the benefits that were hoped for:

- Problems are detected earlier; more pro-active things happen.
- Projects of the system staff have the same weight and visibility as problems reported by users.
- It is easier to communicate the entire appalling scope of the problem to non-staff members.
- When the users try to play "My queue is bigger than yours" they always lose.

In addition, it turns out to have several unexpected benefits:

- People find removing items from the action queue inherently rewarding; trivial jobs that have been waiting for years actually get done.
- The action queue is handled by lots of people; since problems are now obvious to people besides the ones that already know how to fix them, more processes are documented.
- Making complainers has required that things be configured more consistently - programs need to be able to figure things out that people could guess at before.
- Developing complainers leads naturally to developing fixers.
- People who had previously claimed that it was extremely important for them to be told every day what was in the queue and what its priority was decided that maybe they really didn't care all that much after all. Similarly, a number of people who had wanted to see every closed action item realized finally that this information was not useful to them. This

noticeably reduced arguments about the titles of items and the exact relative priorities of things with low absolute priorities. (Are you going to do that when hell freezes over enough to skate on, or not until you can build ice fishing huts?)

### Futures

As originally written, the complainers rely heavily on ITAD's environment. Furthermore, they have no options, no configuration files, and a depressingly small number of comments; changing what they do involves changing the Perl code. This is acceptable for small ones, where the code is rapidly comprehensible, but not for the larger ones, where the intent becomes buried and it is difficult to maintain the rule sets. Every complainer that needs to talk to multiple hosts contains the code for identifying hosts that meet specific criteria and running programs on them, allowing the complainers to either have common bugs that therefore have to be fixed multiple times, or even more fascinatingly to all have different bugs.

I am currently engaged in rewriting the complainers for a less centralized environment (notably, one where someone who is not me needs to be able to maintain them). The rewrite involves a single program which takes a rule file to specify how to find the host list, which ones to run on, and what conditions map to what complaints, looking at files or the output of single command lines. This master complainer will not handle every condition; it's appropriate mostly for checking for full disks, error messages in logs, and the like. It won't replace either badaliases or user-problems, so the architecture of having numerous small programs and the occasional intruding monolith will be maintained.

### Related Software

There are a number of other packages aimed at monitoring networked systems to proactively detect problems, including [3], [8], [5], and [4]. These systems are in general considerably more beautiful than this one, but are also in general aimed at immediate notification (via graphs, beeping computers, or beeping beepers) and are not integrated with trouble ticket systems. This system, while unbeautiful, is highly flexible and requires little or nothing from the machines that are being monitored. Trouble ticket systems that have been documented include [8] and [6] in addition to the previously mentioned [7] and [2].

### Author Information

Elizabeth D. Zwicky is a senior system administrator at Silicon Graphics, where she supports the company's internal system administrators. She prefers to program in languages that begin with a "P" and work for companies that have three-letter

abbreviations containing the letter "S". Reach her via US Mail at Silicon Graphics, Mail Stop 730, 2011 N. Shoreline Boulevard, Mountain View, CA 94043-1389. Reach her electronically as [zwicky@corp.sgi.com](mailto:zwicky@corp.sgi.com).

### References

- [1] John L. Furlani, "Modules: Providing a Flexible User Environment", Proceedings of the Fifth USENIX Large Installation Systems Administration Conference, 1991
- [2] Tinsley Galyean, Trent Hein, and Evi Nemeth, "Trouble-MH: A Work-Queue Management Package for a >3 Ring Circus", Proceedings of the Fourth USENIX Large Installation Systems Administration Workshop, 1990
- [3] Stephen E. Hansen and E. Todd Atkins, "Automated System Monitoring and Notification with Swatch", Proceedings of the Seventh USENIX Systems Administration Conference (LISA VII), 1993
- [4] Darren Hardy and Herb M. Morreale, "buzzerd: Automated Systems Monitoring with Notification in a Network Environment", Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI), 1992
- [5] Richard W. Kint, "SCRAPE (System Configuration, Resource And Process Exception) Monitor", Proceedings of the Fifth USENIX Large Installation Systems Administration Conference, 1991
- [6] David Koblas and Paul M. Moriarty, "PITS: A Request Management System", Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI), 1992
- [7] Bryan McDonald, "QMH: A Problem Tracking System", Proceedings of the World Conference on System Administration and Security, 1992
- [8] James M. Sharp, "Request: A Tool for Training New Sys Admins and Managing Old Ones", Proceedings of the Sixth USENIX Systems Administration Conference (LISA VI), 1992
- [8] Carl Shipley and Chinyow Wang, "Monitoring Activity on a Large Unix Network with perl and Syslogd", Proceedings of the Fifth USENIX Large Installation Systems Administration Conference, 1991
- [9] Wietse Venemaa, "TCP Wrapper: Network Monitoring, Access Control and Booby Traps", Proceedings of the Third USENIX UNIX Security Symposium, 1992

# Managing the Ever-Growing To Do List

*Rémy Evard* – Northeastern University

## ABSTRACT

A system administrator's most important task is managing the list of user requests, work assignments, and active problems. If these items aren't prioritized and handled, issues can be forgotten or delayed, and important problems may go unsolved while immediate yet trivial problems get all the attention. In the best case, one will spend too much time working on the list of tasks instead of working on the tasks themselves.

This paper is an account of our experiences with tackling the problem of keeping track of tasks. We present a software system that we have developed and a methodology for using it to stay on top of the growing list of things to accomplish. We feel that our experiences may be of use to other system administration groups.

## Introduction

Sometimes the day of a system administrator goes something like this: You come in bright and early, planning to finally finish that program you've been working on sporadically for the last month. You make the mistake of checking your mail, and see a pile of seemingly simple problems that have built up over night. About an hour later, the truly simple ones have been solved, and you've pushed the not-so-simple ones off until the afternoon. You pull out your program, but notice more mail has come in. You ignore it and start to code, only to be interrupted by the phone. You help out the poor confused user on the other end while poking around your office looking for materials for an upcoming meeting. Your manager stops you, inquires about your long-term projects, and asks you to check on a problem in the machine room. Fifteen minutes later, after fixing a jammed printer, you make it to the machine room, reboot the server, and check your mail while the server comes up. And so the day goes. Exhausted, you head home, knowing you got a lot done, but not knowing exactly what it was.

The point is that the system administrator's job consists of hundreds of tasks from many sources. Users have requests and questions. Managers assign projects and responsibilities. Problems appear from all over. And, perhaps most importantly, you have your own ideas and goals to accomplish.

It is critical to be able to organize all of these tasks. If they aren't handled in some reasonable fashion, then the simple things are taken care of first, while important (but complex) tasks go undone. Worse, some problems get completely forgotten about. The large blocks of time that are required to concentrate on difficult problems become scarce as interrupts become commonplace. As the list of undone tasks grows (if there is such a list), the overhead for keeping up with it takes more and more time. The problem only worsens as the number of users and the number of administrators grows.

This paper is an account of our experiences with tackling the problem of keeping track of tasks. We present a software system that we have developed and a methodology for using it to stay on top of the growing list of things to accomplish.

## Site Information

The Experimental Systems Group manages the computing environment in the College of Computer Science of Northeastern University, consisting of approximately 350 computers of various types and around 1200 active users. The group is made up of both full-time staff members and student volunteers, totaling an average of 10 people each quarter.

## A Contact Point

The first step in a solution is to create a well-known mechanism for the users to submit requests or problems. Whether or not such a mechanism exists, the problems will find their way to you, one way or another. It is to your advantage to choose what that route is. If you don't, you'll have some users visiting your office, some calling you, some emailing you and your manager, some paging you, and some calling you at home. Regardless of the actual method for reporting problems, the user population should be made aware of its existence and how to use it. By creating a method for reporting problems and telling people to use it, you'll limit the number of sources for problems, and you'll cut down on user confusion.

We use a single email alias for user problems, as do many sites. Users are told, repeatedly, to mail requests to "systems". Everyone on the Systems Group receives the mail. When someone replies, they send a copy of the reply to the list, so that everyone can read it and follow the conversation, should they so desire. Most people on the group filter mail to systems into a specific mail folder, making it easier for them to organize and track.

We use "systems" only for request-related mail. When we send mail to the other members of the group for information or discussion purposes, we use a different alias, which allows us to prioritize the mail differently, and organize user requests in one place.

We've thought several times about having more than one mailing list for the users to use. We could have "systems" for most problems, and "macs" for Macintosh-related problems. We have elected not to do this because, in our primarily student-based environment, we don't believe the users will categorize the mail correctly - if it's a network problem with Macs, where should it go? Furthermore, with just one alias, the instructions are almost simple enough for our constantly changing user population: "send mail to systems if you have a problem."

Training the users to send mail to systems is an ongoing effort. When the user's home directory is first created, they're given a README file that, among other things, tells them to send mail to 'systems' if there is a problem. We state this on our hardcopy documentation and on our news postings. And we say it to users in the hall and on the phone if their problem isn't an emergency. If they send mail to someone directly, we resend the mail to 'systems' and send them a canned response saying that their mail should be sent to 'systems'. If they do it again, their mail takes a little longer to be resent...

We want mail to go to "systems" and not to individuals for several reasons. If the individual is gone, busy, or on vacation, no one else will know about the problem, much less be able to work on it. It's useful for others members of the group to know what's going on. Perhaps most importantly, the users are rarely correct in their guess as to who will work on their problem. When they send mail to systems, we can pick and choose who will work on what, rather than letting the user target a specific individual.

We do handle requests from other sources when the need arises. The phone has an answering machine, and we run a help desk during certain hours for people who don't have accounts or can't use mail. And when someone in the hall has a problem, we'll try to solve it if it will take less than a minute or so.

Using a shared email address has several benefits:

- Users have one place to send things.
- Everyone reads the mail, so people are aware of ongoing issues.
- It's a great learning mechanism for junior members of the Systems Group.
- Nearly everyone can send email, and it's pretty simple, (unlike, for example, posting news or running a graphical interface), so it works for most users.

● Logging the mail for record keeping is simple. Unfortunately, it has some real problems as well:

- Everyone reads the same mail, creating some serious overhead.
- Sometimes two or more people try to answer the same problem, wasting time and sending two (potentially confusing) answers to the user.
- Sometimes people assume someone else will answer, or forget to send a copy of their their reply back to systems.
- Everyone has their own mail queue, so everyone tracks problems to make sure they get solved. This translates into duplicated effort and confusion.
- Requests get lost or forgotten in the mail deluge.

Thus, a shared email address is a step in the right direction, but isn't a complete solution.

### Failed Interim Solutions

The most critical failing of the simple email address was that it didn't keep track of requests. I, as manager of the group, spent enormous amounts of time monitoring the email queue, making sure that things got answered and that people were working on the important problems. I tried keeping lists of tasks that needed to be done on paper tablets, on white boards, and in ASCII files. I was always updating the lists and rechecking them. Those lists weren't easy to keep up-to-date, and weren't easily modifiable by everyone in the group.

In order to try to keep up with the incoming queue in real-time while making progress on long-term projects, we developed the concept of a "hotseat", which was intended to be occupied by a person who read systems mail, handling incoming requests and freeing time for others to work on problems requiring more concentration. This failed badly, because it wasn't possible to assign problems to other people or to record the status of a request. While the person on the hotseat did manage to trap interrupts, the next person on the hotseat spent much time duplicating the effort of the previous person.

For quite some time, we planned on developing a better solution than a simple address to keep track of incoming requests. When it became obvious that manually-maintained lists and hotseat organization weren't helping, and that the overhead in trying to keep track of the requests was substantial, we realized it was past the time to move to an automated tracking system.

### Problem Tracking Systems

Many sites use automated tracking systems to keep track of their tasks. After assessing the way that we worked, we decided we needed a system that:

- Kept a database of unsolved requests.

- Attached a user, a priority, a request date, and a due date to a request.
- Let us assign an owner to a request.
- Allowed us to see the list of requests based on varying criteria.

Essentially, it needed be a tool to organize requests and help us choose what to work on next.

Many possible solutions exist, but we were unable to locate a system that fit our needs. Commercial solutions tend to be very expensive, as most require a powerful data base engine. Existing free implementations didn't quite work either. Some, like Queue-MH and PITS, required that the administrator use a specific mail program or tracking system interface. Others, most of which were based on the

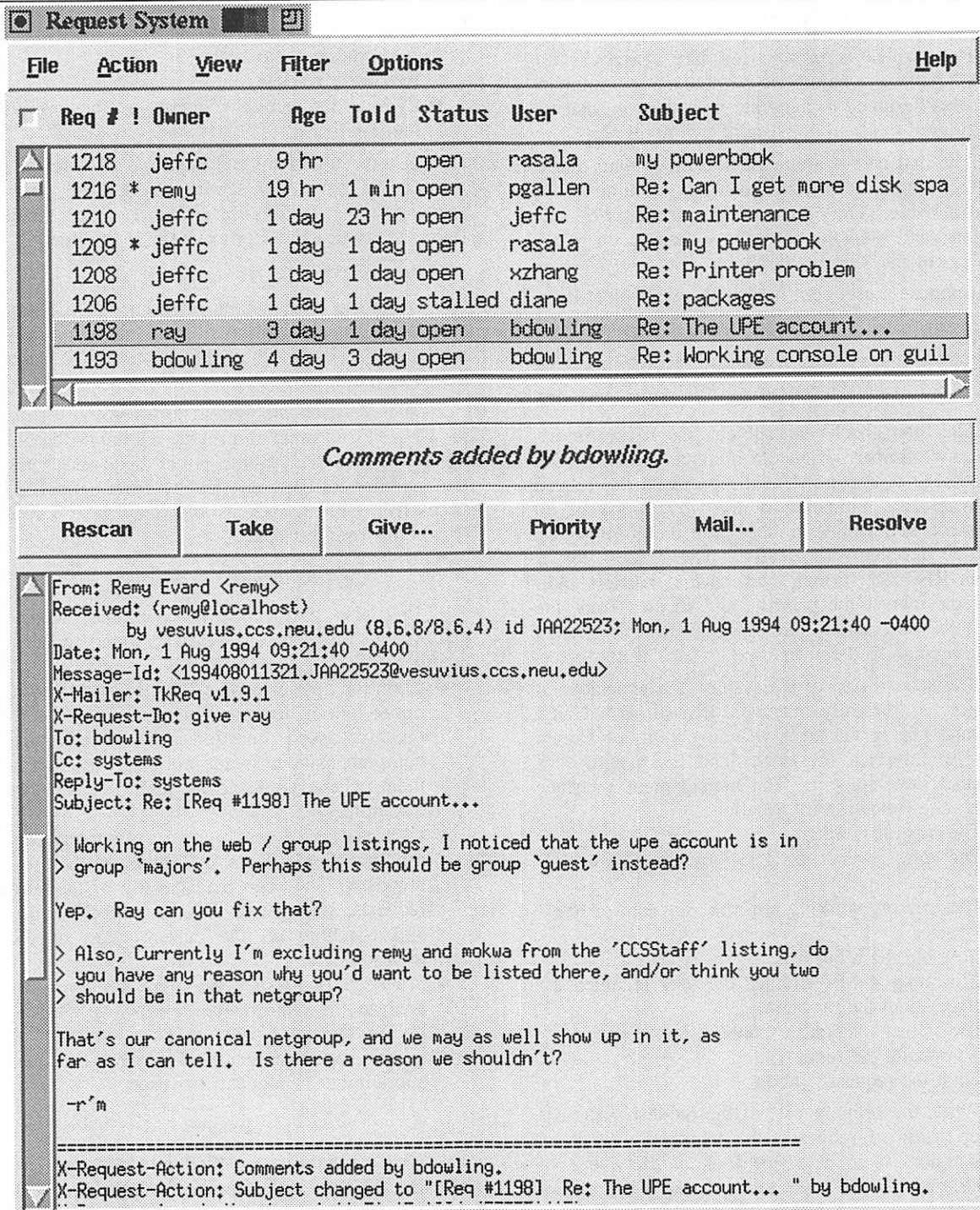


Figure 1: Example management interface screen

UNIX dbm library routines, weren't portable across all the UNIX platforms that we needed to run them on. And others, such as GNATS, just didn't fit our work model. A list of tracking systems that we tried is given in Appendix A. While they didn't fit our needs, they may well be appropriate for yours.

Since we couldn't locate a solution that we thought would work for us, we developed our own.

### Our Solution: Req

Req, which is pronounced like "wreck", not "reek" (although neither is particularly complimentary), was designed to integrate with the way that we already used mail. It consists of two main parts.

The first part is an email filter. All mail that is sent to "systems" is assigned a unique number and stored in a file. The number is inserted into the subject line, and then the message is passed on to the members of the Systems Group.

Suppose a user sent mail with this subject line:

Subject: Help! How do I send mail?

Everyone on the mailing list would receive this mail:

Subject: [Req #1837] Help! How do I send mail?

The filter checks the subject line before inserting a new number. If an old number already exists, the message is appended to the previous file related to that request number. In this way, a log of all mail associated with a number (and therefore with a particular problem) is created. The request log is kept in RFC-822 format, with special headers indicating the owner, the priority, and so on. Thus, the log looks very much like a conventional Internet email message, and can be used as such if necessary.

The second part of the system is a management interface. It currently may be accessed on a UNIX command line or via an X Window tool; see Figure One. An interface for emacs and for Macintoshes are under development. The management interface displays the following information:

- The request number.
- The priority as assigned by the Systems Group.
- The person working on the request, or the "owner".
- How old the request is.
- How long it's been since the user received a reply about the problem.
- The status: "stalled", "open", etc.
- Who made the request.
- What the request is about.

While the primary role of the request system is to keep track of requests, the major use of it by administrators is to help one look at the queue of requests and decide what to work on. The allows one to scan the items that he or she owns or that are currently unowned, choosing which request to work

on based on priority, length of time in the queue, or whim.

Using the interface, an authorized user may:

- Review a request.
- Send mail to the user about the request.
- Take ownership of problems, or assign them to others.
- Prioritize items.
- Merge related items.
- Change their status.
- Add comments to a file.
- Resolve a request.
- Browse the active requests or the resolved requests.

Almost all of these functions can also be performed via email, by sending a message with the request number in the subject line and a command in the message header. For example, if the line:

X-Request-Do: give dave

appears in the header, then the owner of the request will be changed to "dave", and an entry will be made in the log of that request noting the change and who made it.

We designed the system to be as free of policy as we could, in order to provide maximum flexibility while we tested it. Some policy decisions, such as the number of priority levels, were encoded by necessity. Others, such as who may give a request to whom, were left open.

### Additional Features

Req also has a few features that, while not essential to the purpose, we have found to be quite useful.

- Common answers. The X interface to req has options to include and to make files of frequently asked questions. We keep these comprehensive answers in a directory, and use them whenever someone asks one of those questions.
- User access. Users may run an interface to req that shows them the status of their own requests. They can read the log of their own requests, seeing who has been working on it and checking on the current state of their request.
- Statistical analysis. We have programs that analyze the request queue, showing us trends in the requests, such as busy times of the quarter, average length to resolve a request, and number of requests per user.

### Usage

Over time, we've settled into a usage pattern with req. We still tell users to mail their requests to "systems". Each request is assigned a number, entered into the system, and is sent on to the group.

We revived the "hotseat" idea once req was in place. The role of the person on the hotseat is to work with the req interface, keeping an eye on incoming problems and acting as a buffer for the other members of the group. If a new problem can be handled in 15 minutes or less, the person on the hotseat works on it immediately. If it can't be, the hotseat person gives the request to some other member of the systems group, who is notified by mail. The person on hotseat also answers the phone and sits at the help desk during help desk hours. Essentially, this person acts as the only interface to user problems, shielding the other members of the group from interruptions while giving quick feedback to simple user requests. When this person isn't handling interruptions, he or she works on the request queue, making sure that all the problems are owned and making progress on simpler requests. We trade off hotseat duty - each group member is on the hotseat at most one day a week.

The other members of the group work on longer-term projects. They use the req interface to look at their queues, choosing what to work on based on priority. Because the hotseat person is acting as a shield, others can often put in two or three hours of solid work on a problem, rather than being continually interrupted.

As the manager, I occasionally take the time to look over the whole queue, making sure that I agree with the priorities assigned to items, ensuring that progress is being made on the requests, and assigning some jobs to people who have less work than others.

### Observations

Using a problem tracking system has made it considerably easier to prioritize our time and to keep track of requests. Much less time is spent keeping up with the queue, therefore more time is spent making progress on the items themselves.

We haven't lost any user requests since we installed req. We do have problems that have been in the queue for four months, but those are low priority items. The associated users have been made aware that we won't forget about the problem and we'll get to it when the time is right.

About one week into using req, we made the decision to use it to keep track of everything we needed to do, not simply limiting it to user-related requests. So we put our own tasks in the queue, adding, for example, the list of software we wanted to install, a number of hardware repairs, and everything else that we had been keeping track of on other lists. We had thought briefly about using req to manage another mailing list, but that would have meant that we would have had two queues to look at and choose from, not one. This decision has had an extremely positive effect: instead of trying to

remember to do things, we simply send mail to systems, and let req act as our memory device.

While the req system was designed to be relatively policy-free, policies are definitely important. For example, we made the policy early on that anyone could give any request to anyone else. To give a request to someone means that you think they'll do a better job than you will, not that you think they should do it. This has cut down on the political issues related to being on the hotseat and giving incoming requests to your co-workers or your boss.

Our solution is a bit odd, in that group members see both the mail sent to the systems mailing list and the request in the req system. For the most part, we use the mail as a way of watching what's going on and a convenient way of quickly replying to something, while we use req for keeping track of items and deciding priorities. People actively delete the mail in their systems mailbox now, as opposed to keeping it around forever as they did before req. We will probably move to a system where no one gets any systems mail, but we're reluctant to lose the communication and education functions of the mailing list.

### Problems Left

The request system has, for the most part, solved the problems that we had with incoming requests. However, a few problems related to time management still remain.

We need to learn how to manage large-scale projects. The request system is great for keeping track of small items, like a request to install software or fix a printer, but it's not quite appropriate for planning next year's networking infrastructure. When one is in charge of a big project and has lots of small things to do, it's easy to get stuck on the small things while ignoring the big thing. Our current solution is to create a request item for the project and use it to log progress on the project. This may or may not work - we don't know yet.

While we have merged the systems lists into one location, people still have individual to-do lists of their own, including items like meeting schedules, phone calls to return, and so on. We would like to integrate these types of lists into the overall solution, so that it is possible to tell in one glance what one should be focusing on next.

### Critical Points in a Solution

We have built a system that helps us keep track of user requests and systems administrator tasks. With it, we are able to respond quickly to user requests while still putting concentrated time into long-term projects. The important points of our solution include:

- Letting the users know how to submit requests.

- Keeping track of those requests.
- Organizing the requests in ways that allow us to prioritize them.
- Designing policies and procedures to maximize response time and concentration time.

This solution fits our work model, which is based on an email paradigm and a relatively small group of administrators. The solution that works for you may or may not be similar to ours, and should fit your own work model.

### Availability

The req software was designed to be installed outside of our environment. Req is built in C and perl, while Tkreq, the X interface to req, is written in Tk/Tcl. These packages, as well as documentation that goes into much more detail about req than this paper, is available at ftp.ccs.neu.edu in the file /pub/sysadmin.

### Acknowledgements

In a flash of overnight inspiration, Robert Leslie wrote Tkreq, The X interface to the req system. For weeks thereafter, he was badgered into adding features to it.

Other members of the Systems Group, including Lauren Burka, Brian Dowling, Geoff Hulten, Ivan Judson, Shane Kilmon, Dave Kormann, Ray Matthieu, Jim Mokwa, and Matthew Wojcik, were essential in the design phase of Req, and coped with my endless experimentation and questioning once it was operational.

### Author Information

Rémy Evard is the leader of the Experimental Systems Group at Northeastern University, where he has been for two busy years. He received his M.S. in computer science from the University of Oregon in 1992, where he worked as a graduate student systems administrator. His current research interests include distributed virtual environments and automation of systems administration.

### Bibliography

- Tinsley Galyean, Trent Hein, and Evi Nemeth, Trouble-MH, A Work-Queue Management Package for a >3 Ring Circus, in LISA IV, pp 93-96, Colorado Springs, Co, 1990.
- William Howell, Managing In The 90s: Meeting The Challenge, July 1992. Presented at: SANS-I, Washington, D.C., July 1992, UNC-CAUSE, Asheville, NC, October 1992; International Help Desk Conference, Orlando, FL, February 1993; NC Help Desk Chapter, Greensboro, NC, March 1993
- David Koblas & Paul M. Moriarty, PITS: A Request Management System, in LISA VI, pp 197-202, Long Beach, CA, 1992.

RFC 1297, NOC Internal Integrated Trouble Ticket System; Functional Specification Wishlist.

James M. Sharp, Request: A Tool For Training New Sys Admins and Managing Old Ones, in LISA VI, pp. 69-72, Long Beach, CA, 1992.

### Appendix A

This appendix lists the non-commercial problem tracking systems which we discovered or evaluated. They are presented here in the hopes that they may be useful to others. No attempt to compare them has been made, and there are systems that are not on this list that were unaware of or unable to locate.

Most of these tools, as well as comments about them, may be found at ftp.ccs.neu.edu in the file /pub/sysadmin/tracking.

- GNATS, The Gnu Problem Report Management System, available on prep.ai.mit.edu as /pub/gnu/gnats-3.2.tar.gz. GNATS is oriented towards bug report tracking, but can be used for systems administration. GNATS has a Tk/Tcl based front end called tkgnats, which is available in the GNATS contrib directory.
- The NEARnet Trouble Ticket System, available on ftp.near.net as /pub/nearnet-ticket-system-v1.3.tar. It is built on an Informix Relational Database, and uses Embedded-SQL on top of MMDF.
- NETLOG, the JvNCnet trouble ticketing system, is available via anonymous ftp from ftp.jvnc.net as /pub/netlog-tt.tar.Z. It runs on UNIX systems and does not use a database.
- Queue MH, available on ftp.cs.colorado.edu as /pub/sysadmin/utilities/queuemh.tar.Z, is a set of scripts built around the UNIX MH mail system.
- PTS/Xpts, on ftp.x.org as /contrib/pts\* is an X Windows based problem tracking system for both users and administrators.
- Request, a Task Tracking Tool, is on pearl.sl.gov as /pub/request/request2.1.1.tar.Z. It is written in perl and uses dbm libraries, providing an ASCII interface for problem submission and management.
- Requete, available on ftp.crim.ca in /pub/requete-\*.tar.Z is still under development. It has an X interface.

# Speeding Up UNIX Login by Caching the Initial Environment

Carl Hauser – Xerox Palo Alto Research Center

## ABSTRACT

A package scheme helps users manage the environment variables needed by the applications that they use, but imposes a long delay during login while the environment is incrementally constructed. This paper describes an approach to caching the incrementally constructed environment. The mechanism caches different environments for different operating systems and is robust in the face of users' changes to their `.login` files. For the typical PARC user who enables 11 packages at login, caching reduces the time to login from about 30 seconds to about 5 seconds.

## Introduction

The Xerox Palo Alto Research Center (PARC) research community and support staff use various UNIX systems and applications in the course of their daily work. Applications are stored on file servers and maintained for multiple systems using strategies similar to those used in the NIST Depot [2]. Each application is centrally maintained, but every user's computing environment is highly customized. Login scripts set environment variables, configure terminals, and so on, for the applications that the user actually uses. Managing the contents of the environment is burdensome for users who use many applications on many different kinds of systems.

To alleviate some of the burden on users, PARC implemented a Packages scheme similar to the Modules scheme described by Furlani [1]. Each application (actually application version) is stored on a file server in a directory tree structured according to the Packages conventions. The conventions require that every package's directory has a `top/` subdirectory containing a `README` file and C Shell scripts called `bringover` and `enable`. A user executes the `bringover` script once to establish the permanent state of the application in his home directory. Thereafter, the environment in any shell instance can be prepared for the application by executing the `enable` script. The `enable` script at least adds the package's `bin/` directory to the shell's `PATH` environment variable and its `man/` directory to the `MANPATH` environment variable. In general, however, `enable` scripts may affect environment variables in arbitrary ways.

Every user's `.cshrc` file defines aliases implementing `bringover` and `enable` commands taking a package name as an argument. The aliases locate the `top/` subdirectory for the named package, using the support programs of the Packages system, and `source` the appropriate OS-and-package-specific `bringover` or `enable` file. Therefore, before using a package the very first time, a user executes the

command

```
bringover <packagename>
```

once. Thereafter, executing

```
enable <packagename>
```

in a shell instance sets up the environment variables in that shell to use the package. Users typically enable the packages that they use the most with commands in their `.login` files, but they also enable packages interactively, for example, to try out new software. Notice that, unlike Furlani's Modules, our Packages support only the C Shell (`csh`) and other shells that use the C Shell language. The caching techniques described here could be applied to module systems using other shells, but we have not needed to do so for the PARC environment.

## Problem Statement

As was observed in the Modules system, a few seconds are needed to enable a package. This is acceptable when interactively enabling a single package, but it has proven unacceptable when many packages are enabled in the `.login` script. Since our researchers' work on interoperable and distributed systems leads them to login often to various machines, they soon find the delay during login becoming intolerable.

While reducing the time for each enable would be highly desirable, `csh`'s hashing of the directories on the search path with each change of the `PATH` variable imposes a lower bound at which the delay would still be too great. (The problem is compounded by a large directory containing standins for all known executables that many users place at the end of their search paths. The standins help users figure out what package needs to be enabled to provide the command, but with over 6000 entries the directory is very expensive to hash.)

### Approach

Our caching approach is rooted in the observation that the state of the environment immediately after login is almost always the same for a given user on a given kind of machine. It would only be different if the user changed her *.login* file or the system administrators changed the effect of a package's *enable* script. Changing the *.login* file is an infrequent event. Changing an *enable* script is even rarer.

The environment caching mechanism described here reduces the delay during login by setting all environment variables exactly once during login from a file *source'd* from the user's home directory. A separate file is kept for each OS type. The absence of a cache file for an OS or a change to the *.login* file causes each package to be individually enabled so that the environment is correctly initialized. The cache is recomputed asynchronously at each login so there is at most a one-login delay in correcting the cache for a change made to an *enable* script. Thus, the existence of the cache is transparent to the user, excepting only the shorter time it takes to login and the potential to miss a (rare) *enable* script change for one login.

It would be possible to make the use of the cache sensitive to changes in the *enable* scripts by comparing a timestamp in the cache with the timestamp of the *enable* script, but this was rejected for three reasons. First, the additional time required to locate and *stat* the script files would slow down login in the most frequent case—that of no changes. Second, the vulnerability to changes would remain, because *enable* scripts may themselves have dependencies on other files that might change. Thus, users would have to be warned of the potential anomaly anyway. Finally, implementing such a test would further complicate the system. We judged that these negatives outweighed the benefit of a slightly more

sensitive test for cache invalidity. For similar reasons, the benefits of accuracy and simplicity gained by completely recomputing the cache file at each login outweigh the reduction in system load that might be gained by trying to figure out when such a recomputation is really needed.

### Implementation

Environment caching is implemented by a single, shared C Shell script *source'd* from users' *.login* files. To use it, users modify their *.login* files to initialize the shell variable *packages* with a list of the names of packages to be enabled and then *source* the file *.login-shared*. See Figure 1.

*.login-shared* provides the caching implementation. (The script appears in the Appendix should you want to follow along during the discussion.) It is invoked in two ways: as we have seen, it is *source'd* from users' *.login* files; and *.login-shared*, itself, invokes a *nice'd*, background, C shell also executing *.login-shared*. The first of these establishes the environment for the user's current login session, using a cache if one is available, while the other computes a new cache for use the next time the user logs in. (Separate script files implementing these two functions could be used, but having them in a single file is perhaps a bit easier on our administrators.) *.login-shared* determines which of these two things it's supposed to do based on the definedness and value of the environment variable *MAKEENABLECACHE*: if *MAKEENABLECACHE* is undefined, *.login-shared* must construct the user environment and build a new cache in the background; if *MAKEENABLECACHE* is *YES* it should build a new cache; and if *MAKEENABLECACHE* is *NO* it should do nothing (see discussion of *login -p* below).

To construct the user environment, *.login-shared* looks for a cache file (named *.login-enables-*

```
set path = ( /usr/ucb /bin /usr/bin $HOME/bin /etc /usr/etc \
    /usr/parc/bin )
set packages=\
    ( misc lemacs openwin X11R5 Xmisc afs gdb lcd )
if ( -r $HOME/.login-shared ) then
    # .login-shared enables each package listed in $packages
    source $HOME/.login-shared
else
    # normally done in .login-shared for fast enable cacheing
    foreach p ($packages)
        echo -n " $p"
        enable $p
    end
    echo ""
endif
```

Figure 1: Sample *.login* fragment

<platform> by default) and confirms that its *mtime* is later than that of the *.login* file. If so, it *source*'s the cache. As an additional consistency check, cache files are self-checking against the packages list that they implement. If all goes well, the cache file is *source*'d, constructs the environment and returns indicating success in a shell variable. Should either the *mtime* test or the packages list test fail, *.login-shared* takes the slow path of individually enabling each package in *packages*. Finally it sets *MAK-EENABLECACHE* to *NO* and returns to the user's *.login* file.

The *.login-shared* instance that executes in the background receives the list of packages as its arguments. This shell sees a pristine environment in which none of the packages have been enabled. Its initial environment reflects only the contents of the user's *.cshrc* file and the *.login* file prior to its *source*'ing of *.login-shared*. *.login-shared* records this initial environment state in a temporary file and then enables each of its arguments. When it has finished all of them it compares the new environment with the old and produces a cache file containing a *setenv* command for each environment variable that changed or was newly defined. It is beyond the power of simple English to describe the *sed*, *sort*, *uniq*, and *awk* commands that accomplish the comparison between the results of the two *printenv* commands and their combination into a single collection of *setenv* commands, so please refer to the Appendix for the actual code. Figure 2 is an example cache file produced by *.login-shared*.<sup>1</sup>

One obvious thing to worry about is multiple logins occurring in close succession. Care is

required to ensure that the new cache value is correct in this situation. Temporary file names that *.login-shared* creates include the process id of their creator. Furthermore, the cache files are created with temporary names prior to being *mv*'d to the proper place. Since *mv* isn't atomic, theoretically another login proceeding simultaneously could see an inconsistent state. However, should this happen, that login would just take the long path of separately enabling each package, so no real harm would be done.

The implementation supports all of the operating systems supported by the Packages system including Sun Solaris-1 and Solaris-2 for SPARC systems, IBM AIX 3.2 for the RS6000, SGI Irix 4 and Irix 5 for SGI systems, and OSF1 for the DEC Alpha.

### Performance

A sample of 62 PARC Solaris-1 users enable between 3 and 26 packages in their *.login* files. The mean is 12 packages and the median and mode are each 11 packages. (Solaris-1 is the system used by a large majority of PARC UNIX users. One would be hard-pressed to find 26 enable-able packages for any of the other systems.) Recent measurements indicate that to enable 11 packages during login requires 28 to 35 seconds on a typical SparcStation 2 running Solaris-1. Using a cache to get the same effect takes 4 seconds.

### Gotchas

The *.login-shared* file has gone through several releases over the last two years to correct deficiencies of the original design and implementation. Most have been to adapt the script to deal with the different locations of utilities such as *printenv*, *uniq* and *awk* on the various platforms we support.

<sup>1</sup>Figure 2 has been edited to reduce the line lengths. The *setenv* commands are not (and must not be) split over lines in the file.

```
#
if ( $?debugenables ) echo " "
if ( $?debugenables ) echo -n enable cache \
    created Thu Jan 27 14:47:32 PST 1994 by tregonsee
if (" $packages"=="\
    "import-support-1.0 import-support gnu-2.0 sunpro bridge-2.0") \
    then \
    setenv MANPATH '/import/bridge-2.0/man:/local/sunpro/SUNWspro/man:\
/import/gnu-2.0/sparc-sun-solaris2/man:\
/import/import-support-1.0/sparc-sun-solaris2.2/man:/usr/share/man'
    setenv PATH '/import/bridge-2.0/p2:/local/sunpro/SUNWspro/bin:\
/import/gnu-2.0/sparc-sun-solaris2/bin:\
/import/import-support-1.0/sparc-sun-solaris2.2/bin:.\
/sbin:/usr/sbin:/usr/bin:/etc:/usr/ccs/bin:/usr/ucb:/usr/openwin/bin'
    set didenables
endif
```

Figure 2: A small environment cache file

While tedious to correct, such problems are easily predicted in a multi-platform environment. Apart from the locations of the standard commands, no platform-specific customization has been needed, for example, to use different switches or different *awk* or *sed* scripts on the various platforms.

Two more subtle bugs have emerged and been corrected over this time. The first concerns explicit use of the *login -p* command. (Recall that *login -p* passes its caller's environment to the login shell that it creates.) If *.login-shared* is invoked from a shell started with *login -p*, it must not compute a new cache based on the difference between the original environment it sees and the environment created by enabling the listed packages: the original environment already has the packages enabled. This is the purpose of setting *MAKEENABLECACHE* to *NO* in the environment. Since *MAKEENABLECACHE* is inherited by a forked login shell if other environment variables are, *.login-shared* recognizes the use of *login -p* and doesn't compute a new cache.

The other subtlety concerns environment values containing special characters. The *printenv* command does not quote the values in its output, so this has to be taken care of in the *awk* script that converts *printenv* output to *setenv* commands. While not difficult to fix, this bug was not triggered for a long time after the deployment of the caching programs.

Finally, while not directly a problem with environment caching, the improved performance of login has encouraged people to have lots of packages enabled. This has, in turn, pushed them up against *csh*'s 1K limit on the length of the search path. We have had to produce a variant of *csh* supporting paths up to 4K in length for use at PARC.

### Conclusions

A package scheme providing scripts for setting up shell environments can be very useful to a large community using many applications, but both the Modules system and the PARC Package system suffer from the long time it takes to establish the initial environment at login. The environment caching technique described here reduces the time taken by login by about 25 seconds for a typical user at PARC. Since logins are usually not easily overlapped with other work activity, they tend to be particularly disruptive to thought processes. Saving 25 seconds here is seen as more valuable than saving 25 seconds in some other contexts.

The cache validation scheme used is robust enough to immediately implement changes that a user might make to her list of enabled packages, but lags by one login changes that administrators might make to the effect of a package's *enable* script. Most users have found this behavior acceptable. Users who are uncomfortable with this behavior can easily opt out of using the caches by enabling packages

directly in their *.login* files and invoking *.login-shared* without setting *packages*.

### Acknowledgements

PARC's Package system was originally conceived and implemented by Stan Lanning for SunOS 4.1. Jim Foote contributed much of the multiplatform capability. Dale MacDonald currently maintains the Package system and many of the most commonly used packages. Steve Putz provided examples and fixes for the problem of environment values containing special characters, and Mark Verber acquainted me with the Modules work. As always, discussions with Al Demers provided new insights.

### Availability

*.login-shared* is available for anonymous ftp. The URL is <file://ftp.sage.usenix.org/pub/lisa/lisa8/hauser.tar.Z>

### Author Information

Carl Hauser joined the Computer Science Laboratory at the Xerox Palo Alto Research Center ten years ago as a Member of the Research Staff after five years at the IBM San Jose Research Laboratory. He develops language run-time implementations for multi-threaded languages and has a particular affinity for developing caching solutions to performance problems. His 1980 dissertation at Cornell University concerned verification of parallel programs. Reach him by mail at Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304 or via electronic mail at the address [chauser@parc.xerox.com](mailto:chauser@parc.xerox.com)

### References

- [1] Furlani, J. "Modules: Providing a Flexible User Environment." USENIX Large Installation System Administration V Conf. Proceedings, 1991, pp. 141-152. URL: <file://ftp.sage.usenix.org/pub/lisa/lisa5/furlani91-modules.ps>
- [2] Manheimer, K., Warsaw, B., Clark, S., and Rowe, W. "The Depot: A Framework for Sharing Software Installation Across Organization and UNIX Platform Boundaries." USENIX Large Installation System Administration IV Conference Proceedings, 1991, pp. 37-76. The URL is <file://ftp.sage.usenix.org/pub/lisa/lisa4/manheimer90-depot.troff>

## Appendix A: .login-shared

```
#!/bin/csh -f
# .login-shared: Usage in a .login file
# if you want simple enable caching:
#   set packages=(blank-separated-list-of-package-names-to-be-enabled)
#   # if you want enabling to proceed silently define silentenabling
#   #set silentenables=yes
#   if ( -r $HOME/.login-shared ) then
#       # .login-shared enables each package listed in $packages
#       source $HOME/.login-shared
#   else
#       # if things are set up right, this branch should never be executed
#       foreach p ($packages)
#           enable $p
#       end
#   endif
# if you enable different packages in different situations
#   if (situation-1) then
#       set cachename=.login-cache1
#       set packages=(list-of-packages-for-situation-1)
#   else if (situation-2) then
#       set cachename=.login-cache2
#       set packages=(list-of-packages-for-situation-2)
#   else if ...
#   endif
#   if ( -r $HOME/.login-shared ) then
#       # .login-shared enables each package listed in $packages
#       source $HOME/.login-shared
#   else
#       # if things are set up right, this branch should never be executed
#       foreach p ($packages)
#           enable $p
#       end
#   endif
# if you don't want enable caching
#   unset packages
#   if ( -r $HOME/.login-shared ) then
#       # .login-shared enables each package listed in $packages
#       source $HOME/.login-shared
#   endif

# That's the end of the documentation for ordinary users.  What
# follows is detailed documentation of the caching system.

# Create a .login-enables file to be sourced by .login.  The produced
# file contains setenv commands that reflect environment changes made
# by enable files for the listed packages.  The advantage of using
# enable caching is that the environment setting during .login
# processing goes fast, while the hard work of figuring out what the
# enable files do is deferred.  The disadvantage is that the
# environment variables will be set according to the enable files as
# they existed at the previous login--which, it is claimed, is not too
# bad since enable files are slowly changing objects.

# This file is processed twice: once when sourced from .login, once in
# a forked csh.  The environment variable MAKEENABLECACHE controls its
# operation.
```

```

# MAKEENABLECACHE conventions
# == YES: processing in a forked csh; .login-enables should be created.
# == NO: processing was sourced from a .login that itself is
#       executing in an environment where enables have already been done.
# unset: processing was sourced from a .login that itself is
#       executing in an environment where enables have not yet been done.
# To start using enable caching, change your .login file as described
# above; each subsequent login will use an existing .login-enables to
# speed enable processing then create a more up-to-date one for the
# next login.

# Bugs:
#   SunOS4.1.1 dependent?
#   mv is not really atomic.
#   Presumes that the only effect of the enable files is to modify the
#       values of environment variables.

# Don't alter the search path set by .login. Instead, explicitly
# reference the desired utilities depending on the host platform:
set platform = '/import/import-support-1.0/bin/sys-os-type.1'

if ( ! $?MAKEENABLECACHE ) then
  if ( ! $?packages ) then
    # If $packages is not set, .login has not been set up to use
    # this mechanism properly.
    # We might want to give some advice about using caching enables
    # here.
    exit
  else
    # enable import-support-1.0
    set packages = (import-support-1.0 $packages)
    # if ( $?packages ) then
    #   No MAKEENABLECACHE in environment so fork self to make the file.
    #   if ( ! $?cachename ) set cachename = .login-enables
    #   setenv ENABLECACHE $HOME/$cachename-$platform
    #   setenv MAKEENABLECACHE YES
    # Putting the following command in "()"s make messages go to /dev/null
    # instead of cluttering up the console.
    # But first, sleep a bit so as not to get in the way of the login.
    (sleep 15; /bin/nice /bin/csh -f $HOME/.login-shared $packages &)
    # Henceforth, logins that inherit the current environment should
    # not do this again.
    # If there was no .login-enables for this platform, or if the
    # .login-enables is older than the .login file, we're forced to
    # do the enables synchronously
    unset didenables
    if ( -r $ENABLECACHE ) then
      set LSRESULT = '/bin/ls -c -t $HOME/.login $ENABLECACHE'
      if ( "$LSRESULT[1]" == "$ENABLECACHE" ) then
        # sets didenables if the package list matches
        if ( ! $?silentenables ) echo -n "fast enable: $packages"
        source $ENABLECACHE
        if ( ! $?silentenables && ! $?didenables ) echo -n \
          " -- failed; maybe the list changed?"
        echo ""
      endif
      unset LSRESULT
    endif
    if ( ! $?didenables ) then
      if ( ! $?silentenables ) echo -n "enabling:"
      foreach p ($packages)

```

```

        if ( ! $?silentenables ) echo -n " $p"
        enable $p
    end
    if ( ! $?silentenables ) echo ""
endif
unset didenables
unsetenv ENABLECACHE
# endif
setenv MAKEENABLECACHE NO
endif # $?packages = T
else if ( $MAKEENABLECACHE == YES ) then
# This is the forked self. Update the compiled enable file
# $ENABLECACHE.
# Make the compiled file self-validating wrt the package list
echo "#" > $ENABLECACHE.$$
echo 'if ( $?debugenables )' echo "' " ' >> $ENABLECACHE.$$
echo 'if ( $?debugenables )' echo -n 'enable cache created' \
    'date' by 'hostname' >> $ENABLECACHE.$$
echo 'if ( ' "\"$packages\" " == " \"$*\" ' ) then '\
    >> $ENABLECACHE.$$
# Capture the current environment.
# Each line in prefaced with a "b=" to mark it as being
# "before" the enables.
switch ( $platform )
    case mips-sgi-irix4:
    case mips-sgi-irix5:
    case alpha-dec-osf1:
    case rs6000-ibm-aix:
        set PRINTENV = /bin/printenv
        breaksw
    case sparc-sun-solaris1:
    case m68k-sun-solaris1:
    case sparc-sun-solaris2.3:
    case i486-sun-solaris2:
    default:
        set PRINTENV = /usr/ucb/printenv
        breaksw
endsw
$PRINTENV | /bin/sed -e "s/^/b=/ " > /tmp/envIRON$$
# our own definition of enable since user's .cshrc may not execute
# .cshrc-shared in this forked process
# n.b. for LISA VIII readers:
# the following command had to be improperly split
# across lines to fit on paper. Retrieve the
# actual script to assure correctness.
alias enable \
    'source "/import/import-support-1.0/'$platform' \
    /bin/package-file-name enable \!' "'
# Do an enable for each argument.
foreach p ($*)
    enable $p
end
# Capture the resulting environment, adding it to the file created
# before the enables.
# Each line is prefaced with a "a=" to mark it as being
# "after" the enables.
$PRINTENV | /bin/sed -e "s/^/a=/ " >> /tmp/envIRON$$
unset PRINTENV
# Sort the file with all the environment values. The sort is
# carefully designed to bring together lines that define the same

```

```

# environment variable, with the "after" line before the "before"
# line (if there was indeed a "before" value). Note the clever,
# indirect use of the "a=" and "b=" that were added to each line --
# we depend on the fact that "a" comes before "b", and that "=" is
# the field delimiter.
/bin/sort -t= +1 -2 -o /tmp/envIRON$$ /tmp/envIRON$$
# Delete things that didn't change at all. Note how the leading
# "a=" or "b=" are ignored in the comparison -- we depend on the
# fact that the "a=" and "b=" were added to the beginning of the
# line.
set UNIQ = /bin/uniq
if ( $platform == mips-sgi-irix4 ) then
    set UNIQ = /usr/bin/uniq
endif
if ( $platform == mips-sgi-irix5 ) then
    set UNIQ = /usr/bin/uniq
endif
if ( $platform == sparc-sun-solaris1 ) then
    # solaris-1 uniq is (silently) broken on lines longer than
    # 1000 characters
    set UNIQ = /import/textutils/sparc-sun-sunos4.1/bin/uniq
endif
$UNIQ -u +2 /tmp/envIRON$$ /tmp/envIRON.uniq$$
unset UNIQ
# Collect the lines that remain and that come from the "after"
# environment. At the same time, convert the syntax of the lines to
# "setenv" commands. If I were really an awk hacker, I could
# probably have this command do the "uniq" stuff above, too. But
# I'm not.
set AWK = /bin/awk
if ( $platform == mips-sgi-irix4 ) then
    set AWK = /usr/bin/awk
endif
if ( $platform == mips-sgi-irix5 ) then
    set AWK = /usr/bin/awk
endif
/bin/sed "s/'/'\\''/g" /tmp/envIRON.uniq$$ \
| $AWK -F= 'BEGIN {sq = sprintf("%c", 39)} \
$1 == "a" { print " setenv " $2 " " " \
sq substr($0,length($2)+4) sq }' \
>> $ENABLECACHE.$$
unset AWK
echo " set didenables" >> $ENABLECACHE.$$
echo "endif" >> $ENABLECACHE.$$
# As atomically as possible, move the uniquely-named file to the
# standard place.
/bin/mv -f $ENABLECACHE.$$ $ENABLECACHE
# Remove the temporary files.
/bin/rm /tmp/envIRON*$$
endif # $MAKEENABLECACHE = YES
unset platform

```

# The BNR Standard Login (A Login Configuration Manager)

Christopher Rath – Bell-Northern Research Ltd.

## ABSTRACT

Several years ago, Bell-Northern Research wrote a login configuration system for their UNIX workstation user community. While it performed adequately at its release, it was very quickly overcome by the user community's needs. This configuration system, called the BNR Standard Login, was recently re-written. This paper describes its past and present states; making a case for the continued existence of a login configuration manager in BNR's ever-expanding multi-vendor UNIX workstation network.

## Introduction

Early in 1989, Bell-Northern Research (BNR) began to widely deploy UNIX workstations on users' desks. As more workstations were purchased, system administration became more difficult; of particular concern was the distribution and installation of software updates.

As a result of complaints from users, the UNIX release team became concerned with the configuration burden being placed upon users by new releases of the software: when a new version of a tool was released, the user was often required to change their configuration files in order to use the new software. Feedback from users indicated that they did not like having to make the changes, and disseminating these changes was difficult.

In order to overcome these problems, the BNR Standard Login System (hereafter referred to as *the standard login*) was devised. The standard login consisted of a shell script sourced from the user's `.login` or `.profile` file; this shell script configured the user's environment for all supported applications and allowed safe, automatic deployment of configuration changes to the user's environment; since everyone sourced the same central script.

## Standard Login v1.x Components

Version 1.x of the standard login consisted of several files (and a set of files):

- `bnrsetup.[c]sh` – the script which served as the **main** procedure of the standard login. It was sourced from either the user's `.login` or `.profile` file and was responsible for executing each setup script and **evaling** its output.
- `macros` – a list of setup scripts which *might* be resident on the system. Each entry in the file was the absolute path of a script.
- `.bnrnosetup` – a list of the setup scripts which the user did not want the standard login to execute.
- `.bnrsetuplog` – the output log created by the standard login whenever an error occurred.

`.bnrrc` – a mini name space where applications could store user specific application defaults. It was also where users would enter parameters to the setup scripts.

`os.setup` – the first setup script to be executed by the standard login; always the first entry in the `macros` file.

`X11.setup` – the last setup script to be executed; always the last entry in the `macros` file.

`wga.setup`, `pub.setup`, ... – each of the other setup scripts had a corresponding entry in the `macros` file.<sup>1</sup> When run, they emitted the shell code used to configure a user's environment.

## Standard Login v1.x Processing

The standard login, prior to the re-write completed for this project (i.e., version 1.x), did its work as shown in Figure 1, *Standard Login Version 1.x Process Flow*.

As originally implemented, the standard login was run by placing a source command at the bottom of your `.profile` or `.login` file. This would set up the process environment space and then start X windows. Users could add their own customizations through a parameter list stored in their `.bnrrc` file.

The processing was quite straight-forward: each setup script listed in the `macros` file was examined in turn. If the user had requested that the script be skipped, then the standard login went on to the next entry in the `macros` file; otherwise the setup script would be run and any output from the script **eval'd**.

To run the setup script, a parameter list was retrieved from the user's `.bnrrc` file and appended to the command line used to invoke the script. The setup scripts did everything from validating the platform they were running on, to starting daemons; they ranged in size from 28 to 304 executable-lines in length.

<sup>1</sup>The setup scripts were called *setup macros* when the standard login was originally written.

### The .bnrrc Name Space

As part of the background to understanding both the standard login and the problems associated with it, it is necessary to discuss the the .bnrrc name space.

As originally conceived, the .bnrrc file was simply a place to put options which would be passed to the setup scripts as command line parameters. These .bnrrc file entries had the following format:

```
<SetupScriptName> <VariableName>=<String>
```

The standard login developers' initial intent was to provide a facility for users to set variables used by the setup scripts. Thus, allowing the user to customize a setup script's output. However, as the reader has probably noticed, these entries bear a striking resemblance to UNIX *environ*(7) entries.

About six months after the standard login had been released, it occurred to the standard login developers that it might be useful to have a C function which would emulate the `getenv(3)` UNIX library entry; that is, an `ITgetenv()` function which would be used by applications to retrieve their .bnrrc entries directly. So, a library was created; providing functions to add, delete, and retrieve entries from the .bnrrc file. Thus, the .bnrrc became a name space available to any application using the `ITgetenv()` library.

The function prototype for the `ITgetenv()` function is:

```
char *ITgetenv(const char *VariableName,
               const char *ProductTag);
```

The `ITgetenv()` function works as follows:

1. If the requested `<VariableName>` is set in the process environment space, return that value;
2. Then check the .bnrrc file for the requested `<ProductTag>-<VariableName>` pair, return

that value if found;

3. If no variable was located, return `(void *)NULL`.

### The Problem

As more application providers began to utilize the standard login to configure their applications, users began to experience a deterioration in the length of time it took to login. In some extreme cases, login times of over 10 minutes were being reported.

A secondary problem was experienced by the application providers: few of them knew how to write robust shell scripts. As a result, a non-trivial amount of their project time was spent writing, testing and maintaining their piece of the standard login. One common programming error by neophyte shell programmers involved the generation of error messages: a setup script would encounter an error and so output a message; which would then be passed along to the `eval` as though it were a command – causing the standard login to terminate.

These two problems, excessive execution time and non-trivial development, provided a business case for spending a substantial amount of time either re-writing or eliminating the standard login.

### Specific User Concerns

While the aforementioned problems were the most serious, they were by no means the only ones. The issues uncovered during our initial investigation of the standard login included, in order of assigned priority:

1. too slow to execute
2. too easily broken (i.e. not robust enough)
3. not enough user documentation
4. `PATH` environment variable becoming excessively long

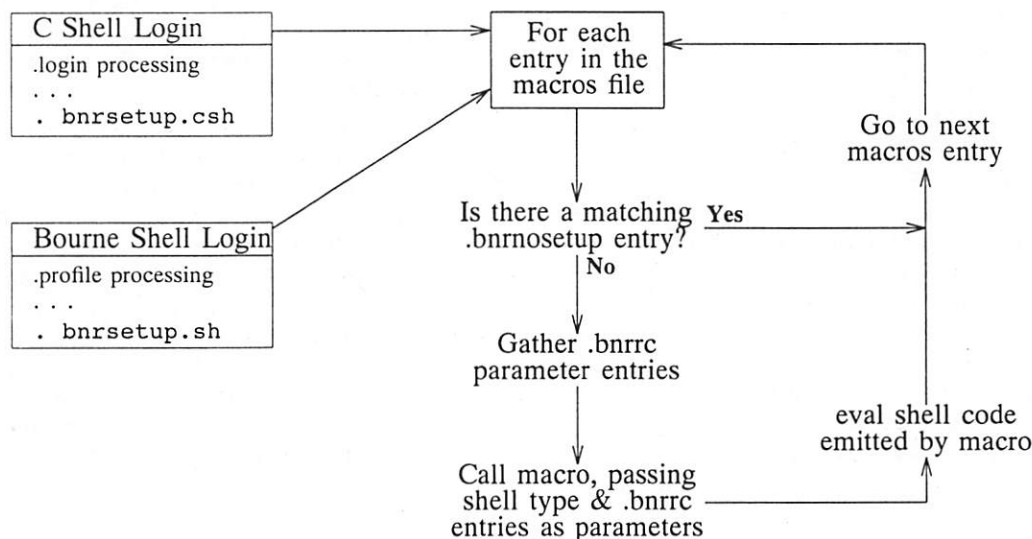


Figure 1: The Standard Login version 1.x process flow

5. no supported location for users' local environmental changes

#### Execution Time

As mentioned above, users on older, and/or heavily loaded hardware, were reporting login times in excess of 10 minutes. With HP VUE becoming the recommended window manager, and as users began to avail themselves of VUE's session save and restore functionality, login times were going to become even longer – even if the standard login portion remained static.

Some timing analysis was performed on the standard login. The results of that timing is shown in Table 1: *Standard Login Version 1.0 Timings*.

As can be seen from Table 1, the standard login spent time performing three basic functions: running the setup scripts, **eval**ing the output from the scripts, and setting up the user's X11 desktop.

The most significant amount of time was spent configuring the user's X11 desktop. However, we are unable to change the amount of time X11 will take to start the user's X-clients. In addition, the user can further degrade this portion of the login by adding to their X11 (VUE or Motif) configuration.

The other significant step was the execution of the `bnrsetup.sh` script and the setup scripts themselves. The user had no direct control over the execution time of these scripts. The 20 seconds spent running the scripts (in this test) was not a large amount of time; but, some of the setup scripts which were not tested took significant amounts of time on their own. One of these, `pde.setup`, is listed in the table.

The `bnrsetup.sh` script itself was a high consumer of time. However, we decided that it wouldn't be sufficient to only optimize that one script. It would be necessary to enhance the execution of the setup scripts themselves too.

It should also be noted that the time to **eval** the setup scripts' output was also not something we could significantly change. However, we could change the time it would take to generate the command stream being sent to the **eval**. We could also attempt to optimize the command stream itself.

#### Not Robust Enough

For a variety of reasons, few of them the fault of the setup scripts' authors, the standard login was very fragile. If an error occurred in either a setup script, or in a user's local configuration files, it would often cause the entire login process to halt prematurely. This meant that the user was either not able to log in, or was left with a partially configured environment.

The user-induced problems we saw included:

- Some `.bnrrc` file Syntax errors would cause the standard login to fail; one example of this was due to a quirk in the standard login's implementation. That quirk allowed C Shell users to use C Shell specific **path** syntax in the `.bnrrc` **PATH** entries. However, as soon as that entry was used by some login shell other than the C Shell (usually because of people copying each others `.bnrrc` files) the standard login would fail.
- User `.bnrrc` **PATH** entries were easily broken; users making changes to their **PATH**

Elapsed Time†	Portion of Time	Processes Executed
78s	100%	Total time‡ to login.
20s	26%	Time to source the <code>bnrsetup.sh</code> script, including execution of the <code>os</code> , <code>wga</code> , <code>ingres</code> , <code>pub</code> , <code>frame</code> , <code>probs</code> and X11 setup scripts.
14s	18%	Time to source only the <code>bnrsetup.sh</code> script, without executing any of the setup scripts themselves.
3s	4%	Time to <b>eval</b> the code output from the setup scripts.
55s	71%	Time to configure the X11 desktop (with <code>mwm</code> , <code>XClock</code> , three <code>XTerms</code> , <code>XBiff</code> , and <code>Emacs</code> ) and any other login tasks.§
~120s	–	Time to run the <code>pde.setup</code> script* (a script not included in our timing tests).

† Unless otherwise noted, all times were generated using the **time** and **date** UNIX commands on an HP360 workstation. Each value is the average of 4 consecutive measurements.

‡ Measured with a stopwatch.

§ This number was obtained by subtracting the setup script execution time (20s) and **eval** time (3s) from the total login time (78s).

\* This time was reported to us by a user as an average time of execution.

Table 1: Standard Login Version 1.0 Timings

variable, via their `.bnrrc` file, often ended up truncating their `PATH`.

#### *No User Documentation*

Very little documentation existed to help users customize their local environments. Some `man` pages were eventually released, but during the course of my interviews I did not encounter a single individual who had seen them. The `man` pages were stereotypically UNIX too: they contained both user and developer information, presented in a minimalist fashion. While this was perfect for the experienced UNIX user, new-comers were lost.<sup>2</sup> Clearly, entry level documentation, maybe in the form of a reference card, was required.

As well, the default `.bnrrc` file installed in a user's home directory at account creation time didn't contain any useful comments. This further frustrated users attempting to make entries in their `.bnrrc` files.

Another interesting documentation problem involved conflicting information. The `man` pages for some of BNR's internal tools told users to make entries in their `.cshrc` or `.profile` files; however, if the user ran the tool without first making the entries, then the tool would prompt the user for the information and write it into the `.bnrrc` file. This eventually resulted in users making conflicting entries in their UNIX configuration files.

#### *Long PATH Variable*

Almost every setup script added a component to the `PATH`. Many of the scripts added more than one new component. In addition, the method recommended to users for re-ordering their `PATH` caused a duplication of components. Thus, making the `PATH` longer than necessary.

#### *User's Environmental Changes*

No supported location, beyond entries in the `.bnrrc` file, was given to users for making changes to their environment. This was compounded by comments placed in the default login files (`.login`, etc.): comments warning users *not* to make any changes to the files. The problem with this is that only a very few environmental changes were possible via the `.bnrrc` file. The reality was that users sometimes needed to make changes to files other than the `.bnrrc` file; C Shell and Korn shell users especially. So, any attempt to make non-`.bnrrc` file changes unsupported was pointless.

Embedded comments to the contrary, no one was actually attempting to mandate that only `.bnrrc` file changes were supported. The comments had been placed in the default files to

discourage unnecessary fiddling. In retrospect, the comments were too strongly worded.

#### *Specific Developer Issues*

While most problems directly impacted end users, some problems also affected software developers and support personnel. Those specific items identified during our initial inquiries were, in order of assigned priority:

1. insufficient documentation
2. lack of shell script writing skills in software development groups
3. default settings didn't provide enough feedback
4. the standard login scripts were not portable between versions of UNIX (HP-UX, SunOS, AIX, DomainOS)

#### *Insufficient Documentation*

Little documentation existed for setup script writers. The only real documentation consisted of the existing setup scripts themselves. This began to cause serious problems because there was no published programming interface (API). So, setup scripts began to use shell variables that had been intended as private `bnrsetup.[c]sh` variables.

#### *Shell Programming Skills*

There is a learning curve associated with every programming language. The UNIX shells are certainly no exception to this rule. However, because UNIX users use the shell languages at the UNIX prompt, new shell programmers typically believe they are already on top of the learning curve. However, experience shows that nothing could be further from the truth.

In the standard login setup scripts a variety of common mistakes were made time and time again. The mistakes were further compounded as new script writers cribbed code from existing scripts. The most common problems experienced were:

- *Reliance upon an unknown `PATH`*; for example, this resulted in commands such as `grep` running some program on the `PATH` other than `/bin/grep`.
- *Non-specific references to program names*; for example, C Shell users with an `rm` alias<sup>3</sup> would often find themselves being prompted about the removal of the old `.bnrsetup-log` file, when the `bnrsetup.csh` script attempted to remove it.
- *Error messages written to the `stdout` file handle*; as a result, error message often ended up in the `eval` command stream, causing the standard login to abort.
- *Indiscriminate use of unquoted variables*; when an unquoted variable was not defined, execution would sometimes fail because a

<sup>2</sup>It was interesting to observe that even when `man` pages were available, the experienced UNIX users went about discovering how the standard login worked by reading the source code.

<sup>3</sup>Where `rm` was aliased to `'rm -i'`.

parameter to a command was now missing.

- *Inconsistent use of `set` and `setenv`*; some of the setup scripts used `set` to change variables instead of `setenv`. This was not a problem when the `PATH` and `path` variables were being manipulated. However, one of the standard login's internal variables<sup>4</sup> was often mis-set in this fashion, sometimes resulting in a setup script's changes to that variable being lost.

#### *Execution Feedback*

The lack of user feedback during the execution of the standard login was of more concern to software support than to the script developers. However, it has been included here since this was a design decision that had been made by the standard login developers.

When users experienced difficulty logging into their systems, they called their local software support team. Invariably, since the standard login's default mode was to output no messages, the user was unable to give software support any information. This increased the time it took software support personnel to solve problems with the standard login; and subsequently reduced users' satisfaction with the standard login.

#### *Script Portability*

Stated simply, complex UNIX shell scripts are *not* portable; that is, a script written for HP-UX will invariably break when run under SunOS. While this has recently begun to change, it is still a real problem and it will be with us for a few years yet.

It is beyond the scope of this paper to discuss this in much detail. However, a couple of examples of non-portable behaviour are given here (to bring a little wisdom to the uninitiated):

- *Determination of platform type*; every vendor provides a different mechanism for identifying their hardware and OS release level to a program.
- *Many UNIX commands are implemented differently*; the list of commands that don't work identically on different platforms ranges from `awk` to `ps`, and `echo` to `grep`. While each of these commands performs the same basic functions on all platforms, the specific command line options available differ.

The non-portable nature of shell scripts meant either that different versions of the setup scripts existed on different platforms, or that each script had been constructed using multiple `if-then-elif...-else`

<sup>4</sup>Script writers often tried to change a variable named `PATH_START` using a `set path_start=` command. Amazingly enough, this sometimes produced a correct result due to an undocumented shell variable with this same name. However, it failed to work as soon as more than one script used this private variable name.

clauses to isolate platform dependent code. Both of these situations caused maintenance headaches.<sup>5</sup>

#### **Possible Solutions**

A number of solutions were examined as alternatives to the original standard login. Each of these possible alternatives was considered as a building block in the final solution; we believed that a number of small or medium sized changes was most likely to result in success. The list of alternatives included, in no particular order:

- re-writing the existing setup scripts in C (or some other compiled language)
- delaying the configuration of an application until its execution-time
- a table (configuration file) driven compiled program
- a make-environment program which would only be run when necessary
- optimizing the existing scripts
- remove the standard login and use the windowing system's configuration system

All of the options considered had one criterion in common: no changes were required to the user's personal configuration files in order to implement the option. We did not consider it generally helpful to make automatic and arbitrary changes to files in a user's home directory.

#### **Re-write Scripts In C**

This option was typically the first suggestion everyone made when we talked about the standard login's speed problems. However, it is very costly: every setup script would have to be re-written to gain any substantial benefit. While none of the scripts do anything very complex, the effort required would still be high. We did not want any of our solutions to make more work for the application developers, so this potential solution was not seriously considered.

#### *Other High Level Languages*

Also considered at this time was re-writing the setup scripts in Perl or TCL. These languages were rejected because they presented the same problem as shell scripts: the developer community we were supporting would have to learn a new, seldom used (for them), language. It must also be pointed out that the standard login, as it existed at the time, did not preclude the use of Perl or TCL scripts. However, none of the setup script providers chose any language other than `sh` or `csh`.

<sup>5</sup>An effort to overcome this problem was undertaken within BNR in 1991-92 (see [And92]). The core standard login scripts were re-written to run on all of the platforms then in use: Apollo HP, IBM, and Sun. This resulted in a doubling of the length of the scripts due to differences in the various vendor's UNIX systems.

One benefit this method had over the others was that it could be integrated into the system over time: developers didn't have to do the re-write by any specific date. This was because the setup scripts would all still be called as stand-alone programs and asked to emit shell code. Each setup script could be updated as the developers had time.

### Delayed Configuration

It would be possible to delay the configuration penalty until execution time by placing the configuration script in a central directory. The script would configure a run-time environment and then call the actual application.

This method has the advantage that any changes to the environment take place within the scripts execution space, and not within the user's. Thus **PATHs** don't grow and grow, and environment variables only required by a specific application are hidden from all other applications.

### ITgetenv Library

As mentioned above, a BNR provided library, called the *ITgetenv library*, was available to application developers. It provided transparent access to the `.bnrrc` file via a call modeled after `getenv(3)`. Applications could use the *ITgetenv* library calls to retrieve user's application defaults at execution time, rather than relying upon the standard login to do it for them.

Two problems existed here: there was no shell script access to the `.bnrrc` name space, and applications were not making judicious use of the *ITgetenv* library. However, both of these deficiencies were correctable.

### Table-Driven C Program

Another option was a table-driven C program<sup>6</sup> which would produce shell code to be `eval`d. This C program would replace the setup scripts with *setup tables* which would, potentially, be more maintainable (by application developers) than either the existing scripts or C programs would be.

This method would require all of the existing scripts to be re-written. However, since each script would be reduced to a simpler configuration file, less effort would be required than to re-write the scripts as C programs.

### Make Environment Program

A make environment program could be written which would check the standard login configuration files and scripts, and generate a command stream and save it in a file if the time-stamp on the command stream was older than any of the standard login's files. The saved command stream would then be `eval`d. If none of the standard login's files

had changed since the command stream had last been generated, the command stream would simply be `eval`d at future logins.

This method would have been the cleanest for the setup script writers, since all of the existing scripts could be left as is. However, we would still have been left with the problem of the setup scripts themselves: they were difficult to debug, were easily broken by changes in the environment, and required expertise in shell programming to be properly written.

In cases where no command stream needs to be generated, this solution would probably have run the fastest. However, problems arise:

- *How would different kinds of logins be handled?* Would multiple saved command streams be required; one for X11 logins, one for telnet, etc. While you could easily handle this by treating a change in login type as a change in the standard login's files, this would have meant slower logins for users who login using more than one method on a regular basis.
- *How would logins work when a home directory is NSF mounted on a different platform?* Do we have still more saved command streams, or do we simply keep re-generating one? See the previous point for further discussion.
- *What if a some application goes away?* How does the make-environment program detect that pieces are now missing? Does it matter that a piece is missing? The pre-processing required to decide whether to regenerate the `eval` stream is now going beyond the capability of the average `make` program.

Too many problems exist with this option to seriously attempt to implement it. The heavily networked BNR environment is quite dynamic. That is why the standard login was written in the first place. Writing a program to maintain a static piece of code doesn't appear to make sense.

### More Lightweight Scripts

The setup scripts were spending a considerable amount of their execution time checking parameters, platform type and OS types. Some of the execution overhead could be eliminated by re-writing the scripts so that they assume more about the environment they run in; and common information could be inherited from the central `bnrsetup.sh` script. This would make them less robust, but that might not be a problem as they are programmatically invoked.

An example of this process duplication was the determination of CPU-type. Almost every one of the scripts spent time (from 15 to 40 lines) determining what kind of CPU they were executing on. It would be much more efficient for the `bnrsetup.sh`

<sup>6</sup>As will be seen later in this paper, the term *table-driven* turned out to be a misnomer.

script to determine this up front and then pass it along to the setup scripts themselves.

While this option would considerably enhance the execution of the smaller, shorter setup scripts. The longer, more problematic scripts would only see a small fraction of their execution time improved by this change.

### Stop Using the Standard Login

One simple way of speeding up logins would be to stop using the standard login altogether. Some user's we spoke with did advocate this solution; especially those who had already stopped using the standard login.

We believed that it was possible to consider this request because HP VUE was about to be deployed. VUE did have facilities within it to configure the user environment. In addition, it offered a session save and restore facility that meant users only had to configure their environment once.

There were two serious difficulties with this solution: it was a vendor specific solution that we would have to find replacements for on non-HP platforms, and it didn't address the needs of non-X11 logins. The latter problem was most serious, since many non-X11 based UNIX sessions were initiated by users each day; **telnet** sessions between machines, dial-in from home, and batch (**cron**) jobs are common examples of these non-X11 logins.

### The Final Solution

We approached the construction of our solution by asking two questions:

1. How can we make the login faster?
2. How can we provide extra functionality required by users and developers?

We chose these questions because they summarized the issues which had initially set us upon this trek. The questions ensured that we would not lose our focus as we went on with the project. At a more fundamental level that focus was *serving our customers*, since customer input is what had initially spurred development of the standard login.

### Enhancing Login Speed

In order to speed up the standard login we proposed two things: replacing the script based system with a configuration file (table) driven C program, and working with developers to delay the configuration of some applications until execution-time.

#### Table Driven C Program

After examining the existing setup scripts, we determined that a small configuration language could be defined which would provide most of the functionality required by the setup script writers. It was also our belief that the functionality that might be missing from this small language would not have any significant impact on the developers of the existing setup scripts.

We designed a small finite choice grammar (see [McG80]) which provided commands to manipulate environment variables, determine a file's type, validate a terminal type, and execute arbitrary UNIX command-lines. In all, nineteen commands were created.

In deciding to re-write the standard login in this manner, we were making a decision to create our own small language instead of using one of the plethora of languages already in existence. The primary factors which influenced our decision were:

- tight integration with the **.bnrrc** name space
- fast parsing and execution of the setup tables

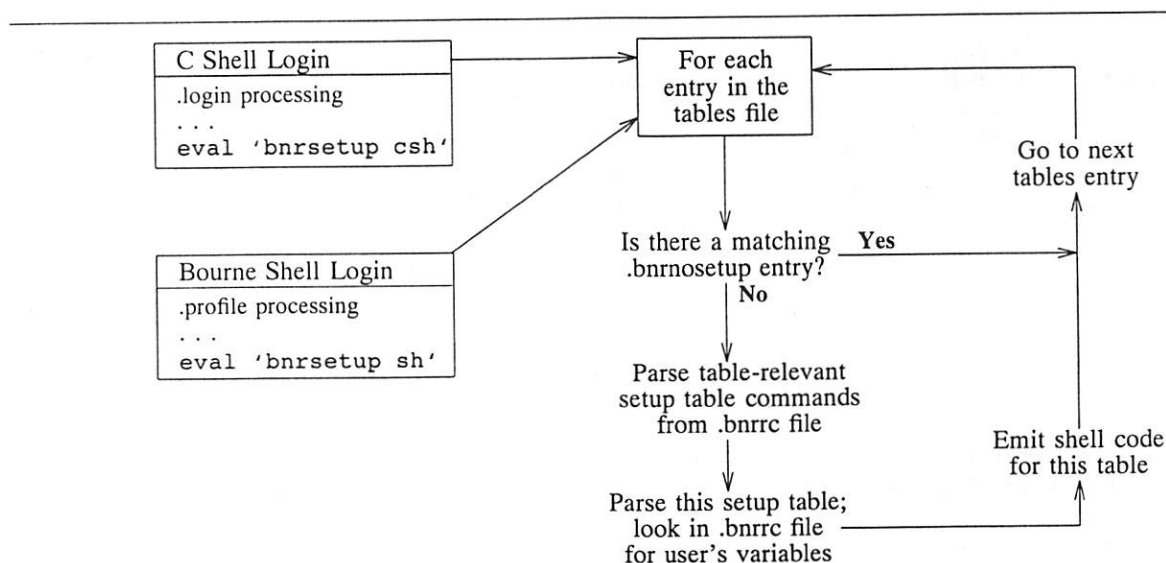


Figure 2: The Standard Login version 2.0 process flow

- to place some limits on what configuration could be done by the standard login
- unrestrained use of all source code and executables used in the standard login, since the standard login is distributed to our development partners around the world

#### Delayed Configuration

The intent of this proposal was twofold: to identify specific tools which would aid developers in delaying configuration, and then to convince developers to modify their applications to support delayed configuration. What this really meant was providing tools for, and convincing developers to use, the `.bnrrc` name space.

Our discussions with developers resulted in the specification of three programs: `rcget`, `rcset`, `rcunset`. These three programs would be used from shell scripts to add, remove and retrieve entries from/to the `.bnrrc` file. A copy of the `rcget`, `rcset`, `rcunset` UNIX man page has been included as an appendix to this paper.

The three programs were designed to be as customizable as possible and to return as much information as possible. As a result of our developer interviews, two application groups shipped early versions of the utilities and their applications in order to make better use of the `.bnrrc` name space before wide distribution of the re-written standard login.

#### Measured Performance

Timing of version 2.0 of the standard login was performed and is shown in Table 1: *Standard Login Version 2.0 Performance Gain*. It is interesting to note that although version 2.0 spent 30% longer evaluating the output code, there is actually 3 times as much code being generated by the standard login (measured in bytes). The particular set of setup tables used in this timing test now generates

3892 bytes (129 commands) of shell commands, compared to version 1.1's 1202 bytes (34 commands).

#### Developer Services

As we discussed above in subsection *Other High Level Languages*, our rejection of that option did place a time-constraint upon the script-providers. To minimize the impact of our re-write, we committed project resources to translate existing setup scripts into new setup tables for *any* script-provider requesting the service.

Over the life of the project, about 15 scripts were re-written on behalf of script-providers; only one of the re-writes took a full day, 3 of the scripts were re-written in under an hour, and the remaining scripts were completed in less than half a day. Once a script had been re-written, the script-provider assumed ownership of, and maintenance responsibility for, the table.

#### Standard Login v2.x Components

The standard login now consists of these files:

`bnrsetup` – the main program which performs all the processing. It is run, and its output is `eval`d, from either the user's `.login` or `.profile` file. `bnrsetup` parses each setup table and emits shell code to a containing `eval` statement.

`tables` – a list of setup tables which *might* be resident on the system. Each entry in the file was the absolute path of a table.

`.bnrnosetup` – a list of the setup tables which the user does not want the standard login to process.

`.bnrsetuplog` – the output log created by the standard login. This file always exists following execution of `bnrsetup`.

Ver. 1.1 Elapsed Time†	Ver. 2.0 Elapsed Time	Version 2.0 Performance	Processes Executed
30.0s	3.0s	10 times faster	Time to run the <code>bnrsetup</code> program, including processing of the <code>os</code> , <code>wga</code> , <code>ingres</code> , <code>frame</code> , <code>probs</code> and <code>X11</code> setup scripts or tables.
8.5s	1.5s	5 times faster	Time to run only the <code>bnrsetup</code> program, without processing any of the setup scripts or tables themselves.
3.0s	4.5s	30% slower	Time to <code>eval</code> the code output from the setup scripts.
33.0s	7.5s	>4 times faster	Total time to <code>eval</code> the output from the <code>bnrsetup</code> program.‡

† All times were generated using the `time` and `date` UNIX commands on an HP360 workstation. Each value is the average of 4 consecutive measurements.

‡ This number was obtained by adding the `bnrsetup` execution time (30.0s and 3.0s, respectively) to the `eval` time (3.0s and 4.5s, respectively).

Table 2: Standard Login Version 2.0 Performance Gain

- `.bnrrc` – a mini name space where applications store user specific application defaults. Users may also place their own setup table commands here; to be executed just prior to the processing of the corresponding setup table.
- `os.setup.table` – the first setup table to be processed by the standard login; always the first entry in the `tables` file.
- `X11.setup` – the last setup table to be processed; always the last entry in the `tables` file.
- `wga.table`, `pub.setup.table`, ... – each of the other setup tables has a corresponding entry in the `tables` file. When processed, they cause shell code to be emitted to configure the user's environment.

### Setup Table Commands

This section contains a description of the setup table commands we created. This list of table commands was generated by examining the setup scripts in use with version 1.x of the standard login. Commands were created to allow all of the functionality required for the setup script writers to re-write their scripts as setup tables.

It is worth noting that one of the commands was incorrectly constructed, `exitIfEqual`. This command should have been implemented as `exitIfNil`. As can be seen in the `FrameMaker` setup table below, the current version of the `exit` command leaves an environment variable set upon exiting from the table. As well, the logic required to use the existing `exit` command is reversed, adding an extra line to the setup table in order to detect the exit condition.

The commands are presented in logical rather than alphabetical order. Note that while setup table commands are not case-sensitive, the command parameters are. Each entry (a.k.a setup table command) in a setup table is terminated with a new line.

`# <comment>` – Begin a comment<sup>7</sup> that continues until a newline is encountered. Note that if the newline is escaped with a backslash (`\`) then the comment continues onto the next line (this is not a recommended practice).

`set <varName> <varValue>` – Set and export the `<varName>`<sup>8</sup> variable.<sup>9</sup> In the C Shell, any shell

<sup>7</sup>`<comment>` is any string terminated by a newline.

<sup>8</sup>`<varName>` is a UNIX environment variable to be assigned a value by the setup table command.

<sup>9</sup>`<varValue>` is a string to be assigned to a variable named elsewhere in the setup table command. One level of expansion is performed on any variables referenced within the `<varValue>` string. Note that C shell colon commands may not be used with variables. Also, empty strings must be doubly quoted to prevent shell syntax errors when the table is processed; for example, `(" ")`. When using variable names within path strings, always enclose the variable name in braces; for example, `'/(some/path:${HOME})'`.

variable with the same name will be `unset` to prevent confusion when `<varName>` is used later in the setup table processing.

`setIfEqual <cmpName1> <cmpName2> <varName> <varValue>` – If the `<cmpName1>`<sup>10</sup> and `<cmpName2>` variables are equal, then assign the new specified value to the `<varName>` variable. See the `set` command for further details. Any of the variable names may refer to the same environment variable. Thus the command, `'setIfEqual test1 test1 test1 val'`, is the same as saying, `'set test1 val'`.

`setIfNil <cmpName> <varName> <varValue>` – Set and export the `<varName>` environment variable only if the `<cmpName>` variable has not yet been set in the environment or the `$HOME/.bnrrc` file. See the `set` command for further details. Note that `<cmpName>` and `<varName>` may both refer to the same variable.

`setLocal <varName> <varValue>` – In the Bourne and Korn shells, set but do not export an environment variable called `<varName>`. In the C shell, set a shell variable called `<varName>`.

`unset <varName>` – Remove an environment variable that was previously set. In the C Shell any shell variable will be `unset` along with the like-named environment variable.

`realize <varName>` – Take the `$HOME/.bnrrc` file variable, `<varName>`, and move it into the setup table's environment. This function is provided to allow the variables found in the `$HOME/.bnrrc` file to be easily modified within the setup table. If `<varName>` is already defined in the environment, or if no such variable exists in the `$HOME/.bnrrc` file for this table, then no action will be taken by this command.

`prefixPath <varName> <varValue>` – Prefix `<varValue>` to the named path environment variable, `<varName>`. See the `setPath` command for further details.

`appendPath <varName> <varValue>` – Append `<varValue>` to the named path environment variable, `<varName>`. See the `setPath` command for further details.

`setPath <varName> <varValue>` – Assign `<varValue>` to the `<varName>` path variable. Before the new path variable is actually sent to the shell, the path contained in `<varName>` is scanned and any duplicate components are removed.

`setPathIfNil <cmpName> <varName> <varValue>` – Set the `<varName>` path environment variable only if the `<cmpName>` variable has not yet

<sup>10</sup>`<cmpName>`, `<cmpName1>` and `<cmpName2>` are variables whose values are to be used in a comparison by the setup table command.

been set in the environment or the \$HOME/.bnrrc file. See the **setPath** command for more details.

**chkFile** <filename> <varName> – Determine the file type<sup>11</sup> and place it in the named environment variable. One of **directory**, **error**, **executable**, **nonexistent** and **regular** will be returned in <varName> by **chkFile**. This command is similar to the UNIX **test(1)** command. Note that any <filename> which is not a directory, an executable file, or a regular file, and which does not cause an error, will be identified as **nonexistent**.

<sup>11</sup><filename> is any valid UNIX filename. One level of expansion is performed on any variables referenced within the <filename> string.

**exitIfEqual** <cmpName1> <cmpName2> – Discontinue processing of the table if the two environment variables have the same value. When this command is used in a \$HOME/.bnrrc file, it only causes processing of the \$HOME/.bnrrc file to stop for that table, not the processing of the entire setup table.

**exec** <command-line> – Run a program in the background.

**exec-fg** <command-line> – Run a program in the foreground.

**execIfNil** <cmpName> <command-line> – Examine the <cmpName> variable; if it has not yet been set in either the environment or the \$HOME/.bnrrc file, then execute the command line in the background.

```
## BNR/NT Setup Table
#####
## /bnr/etc/frame.setup.table - Table to configure FrameMaker.
#####
# Unset the _NoPkg and _OK variables, since we're going to need to
# detect their presence, or lack thereof, later.
unset _NoPkg
unset _OK

###
# Set up to see if we're on a supported platform.
setIfEqual CPU _HP _OK (YES)
setIfEqual CPU _HPPA _OK (YES)
setIfEqual CPU _SUN4 _OK (YES)
###
# If _OK didn't get set then we're not on a supported platform. So,
# output a message to the user saying that we couldn't configure
# frame, and then exit.
execIfNil-fg _OK (echo 'FrameMaker is not supported on this platform.')
execIfNil-fg _OK (echo 'FrameMaker setup failed.')

setIfNil _OK _NoPkg (YES)
exitIfEqual _NoPkg _YES

###
# We're on a supported platform, so do configuration. Set the
# DIR_FRAME variable; always deferring to what the user may have
# entered.
realize DIR_FRAME
setIfNil DIR_FRAME DIR_FRAME (/bnr/3rdparty/frame)
###
# Add the DIR_FRAME directory to the path.
prefixPath PATH_TAIL (${DIR_FRAME}/bin)
###
# Now clean up by getting rid of the DIR_FRAME environment variable;
# and the junk left over from the platform check.
unset DIR_FRAME
unset _OK
unset _NoPkg
###
# End of the table.
```

Figure 3: FrameMaker Setup Table

**execIfNil-fg** <cmpName> <command-line>  
 – Examine the <cmpName> variable; if it has not yet been set in either the environment or the \$HOME/.bnrrc file, then execute the command line in the foreground.

**execIfEqual** <cmpName1> <cmpName2>  
 <command-line> – Compare the two environment variables; if they are equal, then execute the command line in the background.

**execIfEqual-fg** <cmpName1> <cmpName2>  
 <command-line> – Compare the two environment variables; if they are equal, then execute the command line<sup>12</sup> in the foreground.

**configTerm** – Examine the **TERM** variable and verify that the terminal type is found in the terminfo database. If it is not found then prompt the user for a value. If the user simply presses the [return] key at the prompt then use the value set by the \$HOME/.bnrrc file's **DEF\_TERM** variable. If no **DEF\_TERM** variable exists, then assign a default, platform dependent, value.

If the user enters a terminal type that is not found in the terminfo database, re-prompt the user, warning of the error. The user may simply press [return] once again to force the invalid terminal type. **configTerm** will re-prompt two times beyond the initial prompt (at most) if invalid types are entered by the user.

### The FrameMaker Setup Table

A setup table which configures a user's environment to run FrameMaker, as supplied with version 2.0 of the standard login, is shown in Figure 3. Although this is taken from the actual standard login release, it has been shortened slightly for inclusion in this paper.

Note the reverse logic preceding the **exitIfEqual** command, as discussed above.

### Project Success Or Failure

At the outset of this project, we chose as our goal an execution time of less than 15 seconds for an average user. From the results of our own timing tests (shown in table 2), and from actual user reports, we can see that we have reached our 15 second goal. As well, setup scripts did prove to be as easy to re-write as we had expected.

### Conclusions

When I began this project I had never used, nor even seen, the standard login. As an outsider, I viewed its existence with a skeptical eye. However,

<sup>12</sup><command-line> is a valid UNIX, shell independent command line. Also, empty strings must be quoted to prevent shell syntax errors when the table is processed; for example, (" ").

after speaking with many users, and a dozen-odd application development groups within the corporation, my skepticism has vanished.

The standard login is widely used within Bell-Northern Research, and with good reason:

- it provides developers with a reliable mechanism for disseminating changes to their application's environment
- it gives users a platform and shell independent method of customizing their desktop environment
- it provides one configuration interface for all of the core applications; especially important since third party (purchased) applications are often configured by an in-house developer using the standard login

These reasons assure the standard login's continued existence. As we begin to provide corporate-wide support for non-UNIX desktop computing, exactly what the standard login will look like five years from now is anyone's guess. However, we do know that any change will be upwardly compatible; and I believe, that compatibility will attract more users and developers to the standard login.

### Software Availability

The standard login and its components are not available in any form.

### Author Information

Christopher Rath has been working in the industry for a little over 10 years. His first exposure to UNIX was as a Xenix Sysadmin in 1985, and over the years he has done programming, analysis, and sysadmin work on a wide variety of OS's and platforms. He is presently employed by Bell-Northern Research, managing the group responsible for packaging and distributing the corporation's desktop UNIX environment. Christopher's mailing address and telephone number are *Bell-Northern Research Ltd., P.O. Box 3511, Station C, Ottawa, ON, Canada, K1Y 4Y7*, and (613) 765-3141, respectively. He may be reached via email at [crath@bnr.ca](mailto:crath@bnr.ca).

### References

- [And92] Doug Anderson. *Development of The Standard Login Scripts*. Co-op Term Report submitted to manager at end of work term, April 1992.
- [Arn92] Bob Arnold, ASK/Ingres Product Division. *If You've Seen One UNIX, You've Seen Them All*. In *LISA V Conference Proceedings*. USENIX Association, September 1991.
- [jpl93] jpl@allegra.att.com. Finding Commands in a Multi-Vendor Universe. Unpublished email message to the author's manager, July 1993.
- [McC92] Tom McConnell (tmcconne@sedona.intel.com). SUMMARY: POLL: What is in X/shell

files? Unpublished USENET posting to comp.windows.x.apps., August 1992.

[McG80] Andrew D. McGettrick. *The Definition of Programming Languages*, volume 11 of *Cambridge Computer Science Texts*. Cambridge University Press, The Pitt Building, Trumpington St., Cambridge, England, CB2 1RP, first edition, 1980.

[Par93] John Parr. Conversations during the standard login re-write. Principle author of the original standard login system, 1993.

[Sko94] David F. Skoll (dfs@doe.carleton.ca). envv – manipulate environment variables in a shell-independent manner. UNIX man page and source code for Version 1.1 of the application, March 1994.

## NAME

*rcget*, *rcset*, *rcunset* – access the BNR standard login environment space (stdlogin)

## SYNOPSIS

*rcget* [-h] [-p] [-s<string>] <product> <variable>

*rcset* [-h] [-v] <product> <variable> <value>

*rcunset* [-h] [-v] <product> <variable>

## DESCRIPTION

*Rcget*, *rcset*, and *rcunset* provide shell access to the **\$HOME/.bnrrc** file.

*Rcget* checks your local environment space for the <variable> and returns that value if it exists. If the <variable> was not found in the local environment then the **\$HOME/.bnrrc** file is checked. If the <variable> has been defined for the <product>, then that value is returned. If the <variable> has not been defined in either the local environment or the **\$HOME/.bnrrc** file then either a null string is returned, or you are prompted for a value which is returned. See the -p option for further information regarding prompting.

*Rcset* places the specified '<product> <variable> <value>' entry in the **\$HOME/.bnrrc** file. Any existing entry is over-written by *rcset*. If the -v option is specified then *rcset* also checks your local environment for the <variable>, warning you if it exists there.

*Rcunset* removes the specified '<variable> <product>' combination from the **\$HOME/.bnrrc** file. If the -v option is specified then *rcunset* also checks your local environment for the <variable>; warning you if it exists there.

In general, the **.bnrrc** file accessed is the one found in your directory, so that is how it has been referred to here. However, if the **\$HOME** variable isn't set, or if it only contains whitespace, then the **.bnrrc** file accessed will be the one in the current working directory of the process.

## Options

Most options are common to all the utilities. However, see the *SYNOPSIS*, above, to determine which options are supported by each command.

-h prints the utility's usage message.

-p causes the utility to prompt for a <value> if one has not yet been specified in either the local environment or the **\$HOME/.bnrrc** file. You are permitted to respond to the prompt with only a carriage-return; this will cause *rcget* to leave the <variable> unset and a null string to be returned.

-s<string>

changes the prompt string used with the -p option. Note that if the <string> has whitespace embedded within it, then quotation marks are required. For example, -s"**This is a multi word prompt string:**" is specified with double quotes around the entire prompt.

-v causes the utility to check your local environment once the **\$HOME/.bnrrc** file has been updated. The presence of the <variable> in the local environment is treated as an error if the -v option has been specified, and an error message is printed. This option is provided because the local environment always over-rides the **\$HOME/.bnrrc** file.

<product>

names the product to which the <variable> belongs. Note that the product name is only significant within the **\$HOME/.bnrrc** file and not within the local environment. A <product> name can not begin with a '-' character.

<variable>

names the variable the utility is to act upon. Note that within the **\$HOME/.bnrrc** file a

variable name alone does not sufficiently identify a record; a product name is also required, whereas, in the local environment, a variable name alone is sufficient. A *<variable>* name cannot begin with a '-' character.

*<value>*

specifies the value of the named '*<product> <variable>*' combination.

#### ADVANCED NOTES

All of these utilities also support two other optional command line parameters:

##### -dumpmsg

causes the utility to dump its message table to **stdout**. This feature is present so that alternate message tables may be constructed for use with the **-l** option. The message table is dumped in the following format:

*<message-number>* - "*<message text>*"

Any newlines, tabs, or other characters which are embedded within the *<message text>* will appear between the quotation marks.

##### -l*<filename>*

loads the *<filename>* as an alternate message table. This could be done in order to provide messages in a different language, or to meet the specific needs of an application. Use the **-dumpmsg** parameter to determine the default content of the utility's message table.

#### FILES

\$HOME/.bnrrc

#### SEE ALSO

bnrrc(5), itgetenv(3), stdlogin(5)

#### DIAGNOSTICS

The utilities return zero upon successful completion, and a value greater than zero upon failure. The specific codes and meanings are:

- 0 Success.
- 100 An error occurred adding the *<variable>* to the \$HOME/.bnrrc file (or updating an existing entry).
- 105 The *<variable>* is set in the local environment.
- 106 A mandatory argument was missing from the command line.
- 108 A file I/O error occurred while attempting to update the \$HOME/.bnrrc file.
- 109 The usage message has been printed; no other processing was done.

#### WARNINGS

These utilities have no way of changing your local environment space.

#### AUTHOR

Bell-Northern Research

# Exporting Home Directories on Demand to PCs

*David Clear and Alan Ibbetson – University of Kent at Canterbury, UK  
Peter Collinson – Hillside Systems*

## ABSTRACT

Our university has run a UNIX service, both for teaching and research, for over fifteen years. Cost considerations have made us retain timesharing hosts (with X terminals), rather than migrate to desktop workstations. We have come to PCs only in the last five years, and only as poor relations to the main timesharing service, due to a lack of manpower.

Our public PCs do not have hard disks, instead they mount system filestore over NFS. Until recently the only per-user filestore was on local 3½" floppy disks and it was left to the users to back up and manage their own data. As the PC service has gained in importance, the demand for centrally managed filestore has prompted the development of a system to give users access to their own personal secure filestore from any public PC on campus. Moreover, the use of mixed-platform teaching has made it a requirement that a user's PC filestore be visible from their UNIX environment.

This paper describes an implementation of a distributed and relatively secure system for providing NFS based personal filestore on a PC. We use an export-on-demand mechanism. A single home directory is exported to just one PC rather than whole partitions being globally exported. The service has been in use since January 1994.

## Introduction

The University of Kent at Canterbury (UKC) has an active computing population of 8,000 out of a total of 11,000 members. This is high for the UK because all our undergraduates have access to email and News. Also, there is an increasing insistence by the Faculties that assessments be word-processed rather than hand written. We offer both UNIX and IBM PC services. Only a few well-funded researchers have desk-top UNIX workstations, with the bulk of our service running from X terminals into large Sun timesharing servers. Most PCs, especially those in public areas, run DOS/Windows and do not have a hard disk. Instead they obtain application binaries over NFS [1] from per-subnet file servers. Until recently, the only permanent filestore we offered for PCs was floppy disk.

We have always wanted to integrate the UNIX and PC user filestore, so that users see the same home directory from both environments, but we are not aware of even moderately secure methods of exporting UNIX filestore to PCs. We feel that blindly exporting whole disk partitions read/write over NFS to large collections of public access PCs is unacceptable. Our hopes that off the shelf solutions would become available have not been realised. PC distributed filestore, such as NetWare [2] or LANmanager [3], is not based on UNIX servers. On the other hand, UNIX based secure distributed filestore such as the Andrew File System [4, 5] is not available on DOS PCs, despite futures indications from the OSF DCE community. The University of Michigan's IFS

project [6], though offering the potential for integrated filestore, is sponsored research and specific to IBM mainframes.

This paper presents our implementation of an authenticated per-PC NFS export of a user's home directory. Once we could rely on the identity of the user sitting at a PC, it also proved straightforward to offer access to chargeable UNIX resources, such as networked laser printers.

## Public Access to Anonymous PCs

The Computing Service at UKC has historically been based on mainframes, then on minis running UNIX. We succeeded in ignoring PCs until five years ago but the initial slow growth in demand has exploded in the last two years. We are currently responsible for about 1,000 IBM compatible PCs, most of which are in public access rooms spread around the campus. All the PCs are connected to the campus network, which comprises 30 ethernet segments interconnected by IP routers and an FDDI backbone. A site licence arrangement throughout the UK higher education sector for PC-NFS [7] has made it our recommended file sharing and communications package. Although we have no control over the protocols used on local subnets, it is a part of the University's strategic plan that the infrastructure should only carry IP. This reduces the cost of maintaining the campus network and also simplifies its management.

Around 80% of our users see the Computing Service just as their provider of PC services. They

have access to powerful PC applications, including electronic mail, and can launch the PCs into either our central services (X, UNIX, VMS, CWIS) or on to the global Internet via one of our UNIX hosts. We have tried hard to ensure that whenever a user sits in front of one of our PCs the picture they see is familiar and consistent. The key to this consistency has been to make the PCs diskless and have their system files served from one of a handful of UNIX file servers over PC-NFS.

We have gained a number of advantages by avoiding the use of local hard disks. Software updates have become simply a matter of using *rdist* to propagate changes from a master server. This is what gives us a consistent user interface on all campus PCs. The servers export the PC software read-only, to prevent users from accidentally (or otherwise) modifying or damaging system filestore. Viruses are not a significant issue at our site, apart from checking files before mounting them on the servers. The final advantage is that the installation of a new PC only takes a few minutes. This approach is not new, Project Athena [8] drew similar conclusions for a distributed UNIX environment.

As with UNIX workstations, there are disadvantages in running PCs without local hard disks. Data access over the network is slow. This is not as bad as might be imagined because read-only NFS, with personal files on floppy disk, does not suffer the server bottleneck associated with synchronous writes. The migration from DOS to Windows at UKC is slow, since much of our hardware is low on

memory. In future, it seems likely that local hard disks will need to be fitted to all Windows-capable PCs, if only to avoid the NFS writes associated with swap traffic. Currently, we serve up to 100 PCs from a single SPARC 2, although we have introduced some redundancy to avoid a server failure causing widespread disruption. This redundancy is provided via a ROM based boot mechanism.

All the ethernet cards in our PCs are fitted with a commercial boot ROM [9]. When the PC is booted, the ROM broadcasts a BOOTP [10] request for a server. The campus IP routers forward these packets to ensure they reach at least two of the file servers. The servers run *bootpd* and use the PC's ethernet address to indicate, via the BOOTP response, which file the PC should load over TFTP [11]. This file is a DOS disk image and is loaded into memory as RAM drive A. Location specific information in files such as *CONFIG.SYS* is patched into this RAM drive with a vendor supplied utility, which is how DOS learns the PC's IP address, netmask and default gateway.

There are seven general PC-NFS file servers. Which one a particular PC chooses depends on its location on the campus network. Redundancy and workload is controlled to a limited extent by system staff adding or removing entries in */etc/bootptab* on each server. This load sharing, coupled with the unpredictability of user location on campus, makes the PC-NFS system file servers a poor choice for holding per-user files.

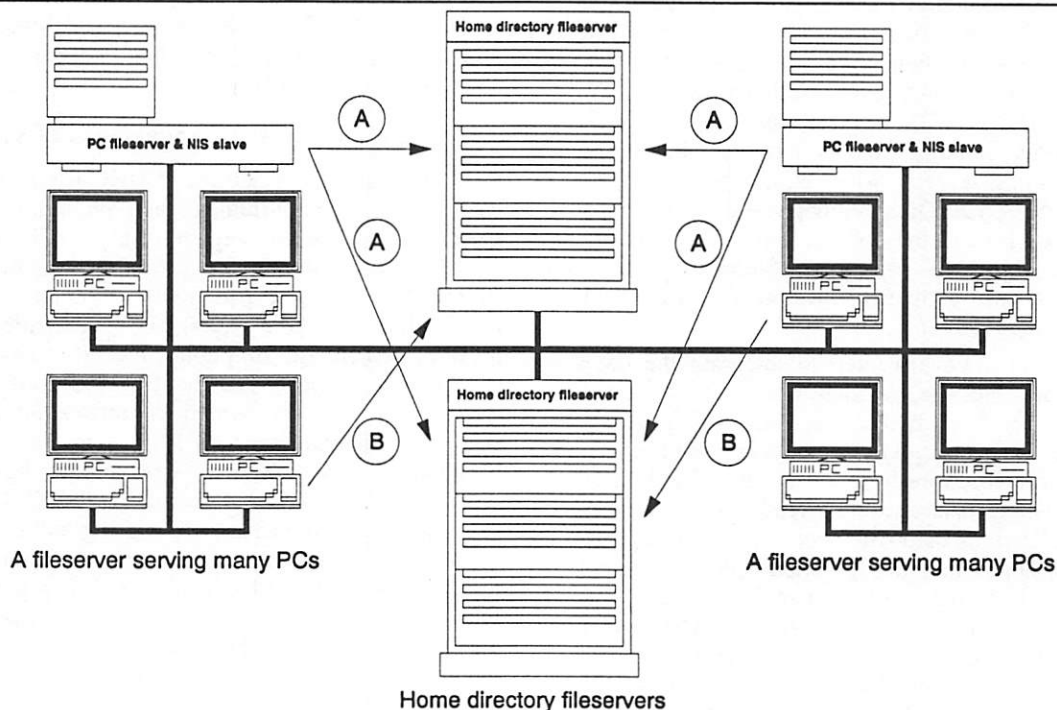


Figure 1: Basic Communicating Host Set

Until a year ago, we sidestepped the issue of personal filestore completely, by not offering anything other than local 3½" floppy disk. Users were left to back up and manage their own data, with sadly predictable results. Our group was therefore commissioned to provide each user with a moderately secure backed-up networked filestore.

### The Problem of Per User Filestore

The use of mixed platform teaching has made it a requirement that a user's PC filestore be visible from their UNIX environment. For a user's UNIX home directory to be accessible to PC-NFS, it must be exported from whichever timesharing host holds that UNIX account. PC-NFS provides a user authentication mechanism for distributed filestore, via the NET NAME command, but it does not provide a secure model of operation. All user filestore must be exported read/write by the server to every PC on campus. If the per-user filestore was located on the PC-NFS system file servers, assuming the geographical problems could be resolved, it might have been just acceptable to abdicate responsibility on the grounds that 'they are only PCs'. However, our plan to coalesce a user's PC and UNIX filestore meant that insecurities could compromise our whole UNIX service.

As an aside, the ease with which uids can be faked to an NFS server from a PC should not be underestimated. It is fairly straightforward to find the PC memory locations holding the uid and gid used by PC-NFS just by a test and compare attack using several legitimate login names. More recently we have found that students have managed to defeat the boot ROM and run a UNIX like system such as Linux from floppy disk. From there, illegal access to any user's private filestore is but a *useradd* and *mount* away.

Figure 1 illustrates the two-level authorisation mechanism which we have implemented in an effort to avoid these obvious security problems.

As can be seen, user home directories are stored on a number of home directory file servers. The arrowed lines represent communications between machines. There are two authorisation levels:

- (A) PC file servers are trusted hosts. Users at the PC must authenticate themselves with the PC-NFS server which will then make a privileged connection to the home directory file server and register the successful authentication from that particular PC.
- (B) PCs are untrusted. They may only communicate with the home directory file server once they have been registered by the PC-NFS server.

### The Initial Menu

A home directory, or for that matter most of the PC applications, are not exported to a PC until the user has logged in.

At the end of the PC boot sequence, DOS runs the AUTOEXEC.BAT script from the RAM disk which was loaded by the boot ROM over TFTP from the PC-NFS server. This script offers the user a menu of just three items: start a telnet session to a campus host, register as a new user, or log in and mount a home directory.

Users are not required to authenticate themselves to the PC service if all they wish to do is use the PC as a dumb terminal. They will connect to a local timesharing host which will enforce its own access controls. Off-site telnet is only available to authenticated users, so we have some hope of tracing malefactors.

The details of new user registration are outside the scope of this paper. We use a pre-registration mechanism for students which is based on information supplied by our central administration just before the start of each academic year. Users collect their login names and choose a password by entering their surname and student number, which is supposedly secret, at least at the beginning of the year. Similar procedures seem to be quite common in the USA, for example [12], but the majority of UK universities still use a paper based system. Our experience is that automation becomes mandatory once all newly arriving students are registered for IT services, especially email.

The third menu item allows the PC user to log in and mount their home directory. The only trusted host known to the PC at this point is the local PC-NFS file server. Each of these servers needs access to the same password file. They also need to know, for each user, which host holds their UNIX account, as well as their home directory pathname. The initial implementation has used standard building blocks as much as possible. In particular we chose the Network Information Services (NIS) [13] for password support because implementations are provided by Sun for both UNIX and PC-NFS.

### Network Information Services

The Network Information Services, formerly called Yellow Pages, are well documented by Sun, but a brief outline is presented here for the uninitiated. The examples are chosen to focus on the specifics of our PC home directory service.

The smallest NIS entities are the key and the data. The key/data pair form the smallest addressable object, the table entry. Table entry keys are unique within a table. Multiple tables are grouped together to form domains. Therefore table names are unique within a domain and domain names are unique across a LAN: Figure 2 illustrates the model.

The highlighted entry is the *ajm1* entry from the *passwd.byname* table of the *comp-pchome* domain.

A NIS server for a domain receives its data from the domain's NIS master. There is one NIS master per domain and any number of NIS slave servers, providing resilience to failure. When a client needs to use the NIS database it broadcasts for a server. Whichever server responds first will be used and the client need not distinguish between a master and slave server.

On the server, the directory */var/yp* is the root for all NIS data. The data for the domain 'dom' is stored under */var/yp/dom* in *ndbm* format files.

NIS tables are transferred (pushed) from the master server to all the slave servers by the *yppush* command. This is required for each table to be updated, four in our case. The mechanism is moderately secure since the slave servers know the identity of the master server.

From a UNIX standpoint, NIS comprises four processes:

- *ypserv* – the NIS server daemon
- *ypbind* – the NIS binding daemon
- *yppush* – the NIS update program
- *libc* – the NIS programmer's interface

Within a NIS server there is one daemon process which handles both requests from clients and the updating of its own information from the master server. This daemon process is called *ypserv*.

When a client request enters *ypserv* for processing, more than a simple table lookup takes place. A table can be created with a security option, by specifying the *-s* flag to *makedbm* when building the table. This means that *ypserv* will only answer requests from clients with low numbered IP ports. It can be used for crude protection of a password table,

for example. The security is easily subverted from the PC world, but it requires knowledge of the RPC programming library. The same security model is used in our PC Home Directory System and is thus an area for improvement.

Another feature of *ypserv* is the ability to forward host table requests to the Domain Name Service [14]. When invoked with the *-d* flag, or when the host tables are built with the *-b* flag to *makedbm*, *ypserv* takes all requests to the *hosts.byname* and *hosts.byaddr* tables and passes them to the DNS for resolution. We use this option at UKC. Smaller sites could equally well elect to use the explicit host tables and not use the DNS.

To look up an entry in a NIS table, a client program must make a connection to a NIS server. It is possible to broadcast for a NIS server but for every process on a client to do this would be inefficient. A level of indirection is therefore added.

A client process talks to a local daemon called *ypbind* to find the network address of a NIS server for the particular domain. *ypbind* maintains a list of domains and servers so that, once it has received a reply to its own broadcast for a NIS server, it can cache and reply instantly to other client process requests. On single process operating systems such as PCs running DOS, the functions of *ypbind* are provided transparently in the NIS library.

### PC Home Directory System Components

Our site has made extensive use of the UNIX group id for many years. We place users in groups depending on their affiliation within the University. For example, *elt* is an undergraduate in Electronics ('t' for teaching) and *elr* is a researcher. For historical reasons, Computing Laboratory staff are in group *cur*.

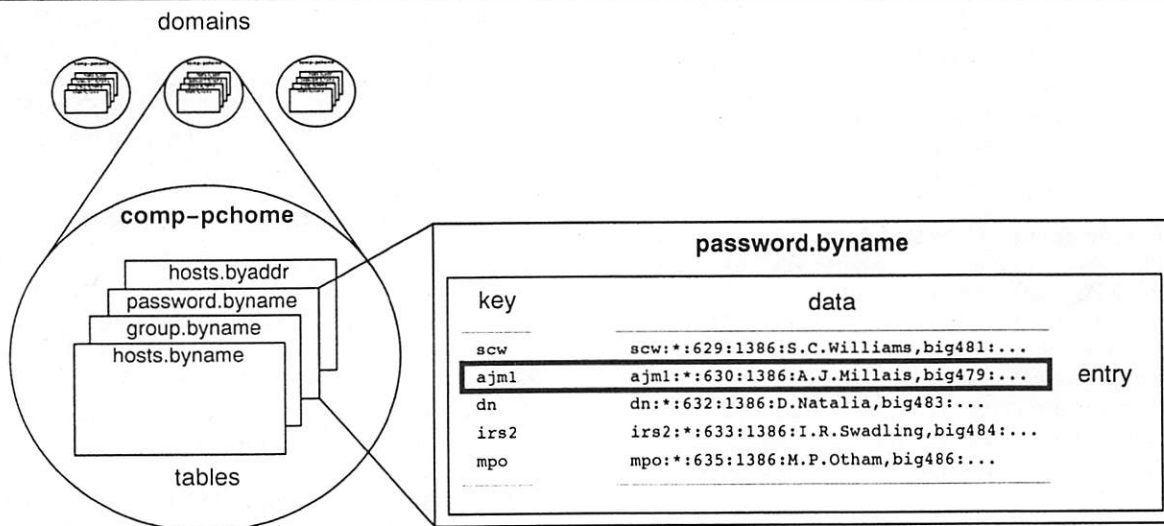


Figure 2: The NIS Namespace Model

User home directories have a consistent structure across all machines. An undergraduate called *fred* in Electronics will find their files in `/home/elt/fred`. Symbolic links at the group level are used to manage this hierarchy. All files in the *elt* group are placed in the same partition for ease of administration, it also aids with peer pressure when partitions fill up.

### The NIS Configuration in the PC Home Directory System

We use the NIS domain `comp-pchome`, which all our public PCs use to resolve host names, group names, services, and networks. The password file, which needs to be accessed by all PC-NFS servers, is available in the standard `passwd.byname` and `passwd.byuid` NIS tables in the `comp-pcuser` domain. Every PC user has an entry in these tables, which contain their password for the PC home directory system and their group id. However, it should be stressed that users do not ordinarily appear in `/etc/passwd` on the PC-NFS filesystems. They do not have a UNIX account there.

There is one other NIS table of interest to the system, the *group.home* table. It performs the mapping from group names to host and path names described above, and tells the system where to look for a particular group's home directories. For example, if it contained the entry:

`cur` maps to `stork:/home/cur` and the user *dac*, who is in group *cur*, logs in, the system will locate *dac*'s home directory as `/home/cur/dac` on host *stork*.

The NIS tables are mastered on the same host as the *rdist* master for PC applications binaries and all other PC filesystems are configured as slave NIS servers. This means that a PC uses the same filesystem for application binaries, PC-NFS authentication and printing, and NIS serving.

### The Faculty Home Directory Fileservers

User home directories are distributed between four SPARCcenter 1000 servers. They support both a general UNIX timesharing service (primarily to X terminals) and PC home directory filesysteming. Filestore on a particular filesystem is allocated to users by group which, as closely as possible, maps to a faculty.

If the *group.home* NIS table points to a user's UNIX home directory, the PC and UNIX filestore is integrated. At the time of writing, we have not yet taken this final step. Each of our users has two directories on one of the SPARC1000s. There is a PC home directory under `/pchome`, pointed to by the *group.home* table, and a UNIX home directory under `/home`, pointed to by `/etc/passwd`. Their PC home directory is visible from their UNIX account but export restrictions protect their UNIX home directory from the PCs. We hope this is not just natural British conservatism. It also seems good

engineering practice to have an extended test period before making a change which will add so much extra functionality that it cannot be withdrawn. We will be throwing the big switch real soon now.

### The PC-NFS Daemon *rpc.pcnfsd*

The PC-NFS authentication and printer daemon *rpc.pcnfsd* was originally supplied by SunSoft. Its functions cover:

- Authenticating usernames and passwords.
- Submitting print jobs to the local PC-NFS server.
- Cancelling print jobs.
- Reporting printer queues.

We modified the daemon to add the functions:

- Registering authenticated PCs with the home directory filesystems.
- Submitting print jobs to a user's home directory filesystem.

The change for PC printing is needed to allow us to charge for laser output. Users do not have UNIX accounts on the PC-NFS filesystems, so instead their print jobs are submitted from their home directory server.

The details of PC printing are straightforward and will not be discussed further here. Instead, we will concentrate on the mechanics of PC authentication and registration with a home directory filesystem.

The user types a login name and a password into a locally written login program on their PC. Spawning the standard PC-NFS command

`NET NAME login password`

sends this information to *rpc.pcnfsd* on the local filesystem. This is far less secure than a Kerberos [15] based mechanism. Although plaintext passwords are sent over the network, it is no worse than our standard telnet service. We are migrating towards UTP and managed hubs to reduce our vulnerability to eavesdropping attacks. The command can only return a success or failure indication, no text or result codes can be passed back, which limits the amount of error reporting which can be done.

The local filesystem is used as the NFS authentication host as it is geographically close. Until users have identified themselves, their home directory server is unknown. The PC-NFS daemon uses NIS for password lookup. If the password is valid, NIS is consulted again to determine, based on the user's group id, which home directory host serves their files. A privileged connection is then made to our *pcshared* service (share is the Solaris 2 term for export) on that host and the user is registered as being correctly authenticated on the PC at which they are sitting. A successful user authentication response is then returned to the PC.

At this point the *NET NAME* command returns control to the client login program. If the login has been accepted, the client makes a connection to

*pcshared* on the home directory server and requests the user's home directory be exported. The export is made *only* to the client PC. If the export cannot be made or *pcshared* does not believe the client PC is authorised, an error message is returned and displayed on screen by the client login program. If all goes well, the client mounts the user's home directory using the PC-NFS command:

```
NET USE H: host:path
```

When this command succeeds, the logging in process is complete.

In order not to tie up *rpc.pcnfsd*, which is single threaded, the connection to *pcshared* is only given 10 seconds to succeed. If the time expires, a failure indication cannot be sent to the client PC. It would cause our invocation of *NET NAME* to emit a 'password incorrect' message, which would confuse the user. Instead, a success indication is returned to the PC. On attempting to request the export, the user will receive a 'share request refused' message from *pcshared*. This message will be displayed as a general error although it might be wiser to retry at this point. There is room for improvement generally in the error messages seen by users.

### The Share Daemon *pcshared*

Our locally written *pcshared* daemon resides on the home directory filesystems. It responds to authorisation messages from trusted PC-NFS servers and subsequently to export requests from previously authorised client PCs.

PC-NFS authentication servers have a trusted status with the *pcshared* daemon. Such trusted hosts may register client PCs, specifying a user and a home directory path, using a telnet-like protocol on a low numbered TCP port. An example authorisation message sent from a PC-NFS server for user *dac* sitting at PC *pcpra* is:

```
authorise pcpra dac /home/cur/dac
```

Once the user authentication is complete, the PC client login program uses NIS to select the host serving the user's home directory and makes a connection to *pcshared*. The PC requests that the user's home directory be exported to it, thus:

```
share dac /home/cur/dac
```

As the client was previously registered by the trusted PC-NFS server, the request is accepted and the export is performed. The full dialogue is shown in Figure 3.

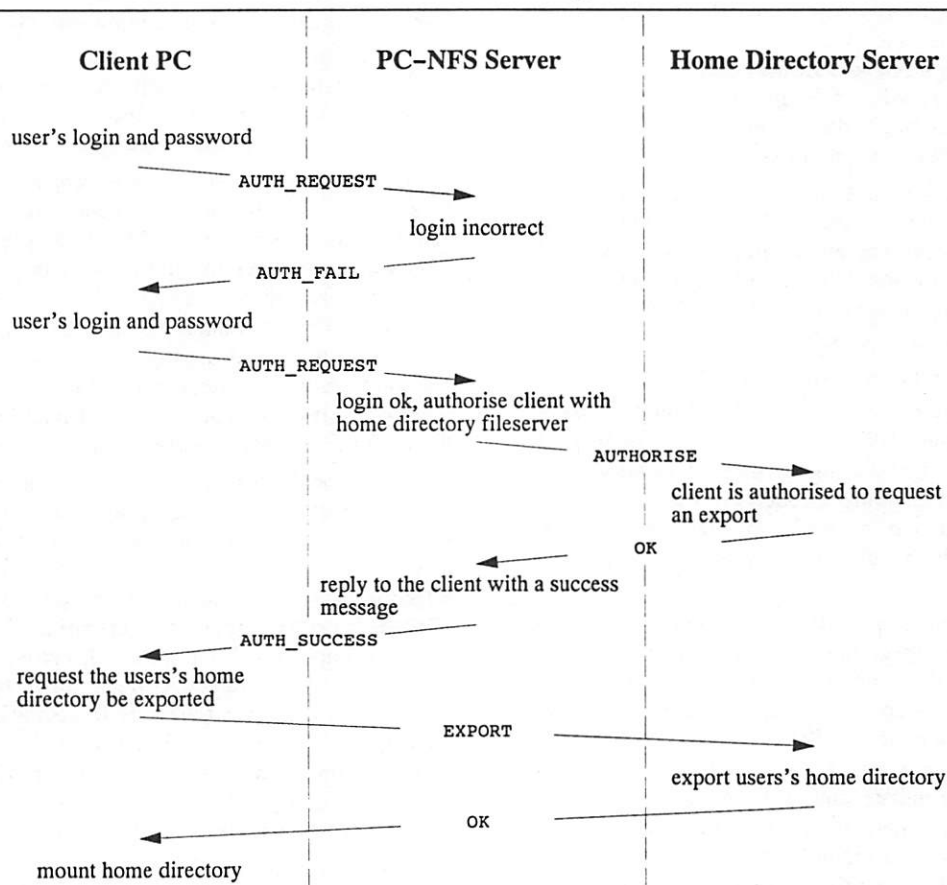


Figure 3: The Logging In Dialogue

The figure shows two attempts to log in. The first is with an incorrect password and the second succeeds. At first sight it might seem more natural for *rpc.pcnfsd* to directly request the home directory server to export the user's home directory. There are several reasons why the above system is used:

- The home directory server daemon, *pcshared*, will only listen to trusted hosts in the first instance. Only PC filesystems are trusted as they are secure UNIX machines with no access by users. Thus, the fileserver must talk directly to *pcshared*.
- The mechanics of exporting a directory are such that it can be a time consuming process. PC-NFS authentication is based on RPC/UDP datagrams. If *rpc.pcnfsd* does not respond quickly enough then another request will be sent by the client PC's RPC network layer. Thus *rpc.pcnfsd* must not do anything that takes too much time.
- To get around the RPC timeout problem, it might be thought that *rpc.pcnfsd* could fork and let a child process request the export from *pcshared*. The parent would then respond with a success message to the client. This would not work. The client PC needs to know when the export has completed, otherwise it will try to mount an unexported directory.
- By increasing the involvement of the PC client login program, more error conditions can be reported to the user. The simple success/fail response codes returned by *rpc.pcnfsd* are not suitable for the wide range of possible failures.

Although PC-NFS servers are trusted, a configuration file on the home directory fileserver limits which filesystems may be exported.

#### Logging out

When the user finishes a PC session they are expected to run our *LOGOUT.EXE* command, which sends an *unshare* request to *pcshared* on the home directory server to remove the export. Although we try hard to remind them, users are apt to walk away without logging out – sometimes they have no choice if the PC crashes. We partially counter this by scanning the entire list of exports when a new user logs in, and cleaning up previous entries for that PC. However, this only works if both users have their home directory on the same server.

#### Password Changing

A user may change their password by running *PASSWD.EXE*. This command determines the host-name of the master NIS server for the comp-pchome domain and makes a connection to the *nispwupdated* daemon on that machine.

Once the connection to *nispwupdated* is established, it sends the client a random single byte.

This is used to very lightly encrypt the data stream. The goal is not to make the password change dialogue secure (only a Kerberos style system could do that), rather to make it harder to read for the casual listener.

After the random byte has been sent, the user is prompted for their username and password which is then sent, lightly disfigured, to *nispwupdated* for checking. Once a successful match is made, the user is prompted for a new password. If it is deemed to be acceptable [16], it is put into the password file. After the password dbm files have been updated, they are *yppush*'ed to the NIS slave servers by the daemon.

#### Disk Quota Reporting

Users may find out their disk quota by using the *QUOTA.EXE* command, which connects to the *pcquotad* service on their home directory server. Disk quotas are not considered sensitive information, so the client need only send the user's login name to *pcquotad* which will respond with the user's quota data. The client program then nicely formats this data and displays it on screen.

#### Experiences to Date

We presently have 1750 users using 8 Gb of filestore. There are 4 SPARCcenter 1000s, which serve home directories to a community of over 1,000 PCs as well as providing a general UNIX service, and 7 PC-NFS servers. In our hands, *yppush* has proved relatively unreliable. It is invoked by the NIS master to propagate the maps to the slaves every time a user changes their password. Whether the RPC mechanism is inherently unreliable or race conditions arise when several users change their passwords at once is unclear. The observed effect is that, after changing their password, a user may be able to log in only at PCs served by a subset of the slave PC-NFS servers. Running *yppush* a second time by hand usually clears the problem.

#### Future Directions

The current implementation of the PC Home Directory System as described has several failings. With hindsight, we would make improvements in the following areas: serving passwords to the network in a secure way, making a cleaner login sequence, and providing more resilience on a server crash.

#### Network Password Serving

The use of NIS to serve passwords has the disadvantage that it is not secure. Access may be gained to the encrypted password strings by anyone with socket and RPC programming knowledge. Another problem with NIS is that, when a single password is updated, the entire password table is shipped to all NIS slaves. As some filesystems sit on the same subnets as public PCs, the password file is vulnerable to *netwatch* style programs.

A way forward would be to use a bespoke password file transport mechanism. This would take a similar form to how NIS does things (with a master host) but with the following differences:

- TCP connections could be used instead of UDP as used by NIS. We suspect that this would make the transfer of passwords more robust.
- Only password file differences would be sent out to the password server slaves. This removes the ability for a user to trigger the movement of the entire password file.
- The password file transfer could be encrypted, to thwart eavesdroppers.

The use of such a method would be compatible with the current system as only *rpc.pcnfsd* uses NIS to look up user passwords.

#### A Cleaner Login Sequence

The *rpc.pcnfsd* interface for login authentication is not particularly good at reporting errors. A possible alternative to this would be to remove the user authentication and export requests from *rpc.pcnfsd* altogether and build a new login daemon to handle that. The difficulty is that PC-NFS on the client PC must run the NET NAME program to authenticate the user. A new model which could accommodate this could be:

1. The user on the client PC runs LOGIN.EXE which makes a connection to the login daemon on the PC-NFS fileserver.
2. The user then has a dialogue similar to that of *nispwupdated*, with lightly encrypted network traffic, to pass across the username and password.
3. The login daemon checks the username and password and, if incorrect reports it as such.
4. If the password is correct the login daemon has the user's home directory exported by talking to a *pcshared* style process on the user's home directory host. Notice that, as this is a fully synchronous system, the trusted host is free to ask for the export to be performed itself.
5. Once exported, the login is complete so the login daemon registers that the user has logged in with *rpc.pcnfsd*. Then it returns a success code to the client.
6. The client runs NET NAME to perform the mandatory PC-NFS logging in functions. *rpc.pcnfsd* can simply reply with a positive return code as it has been told that the user has already correctly logged in. No password need be given or checked.

The advantages of this system are that

- The home directory export request is made to the home directory server by the trusted PC-NFS host. There is no authorisation step.
- There is a fully synchronous error reporting path. This makes error trapping and reporting

trivial. It removes the need for the Message Send Protocol [17], which we use (somewhat untidily) at present.

- At no point is *rpc.pcnfsd* tied up, thus leaving it to handle print requests as normal.

#### Persistent Exports

At the moment, if a home directory server reboots for any reason, the export table is lost. To the user this would be seen as an 'access denied' message from the fileserver once it came back to life.

This could be better handled by having *pcshared* record the exports it has made in a private file which, on startup, it could read to pre-configure itself. If a fileserver were to reboot the home directories would be re-exported and the users would not be forced to log back in.

#### Availability

The PC home directory software is available by anonymous ftp from <ftp.ukc.ac.uk> in directory `pub/pchome`.

#### Author Information

David Clear obtained his bachelors degree in Computer Science from the University of Kent in 1990. Since then he has worked for the Computing Service Division, specialising in UNIX systems support. Reach him at the Systems Group, Computing Laboratory, University of Kent, Canterbury CT2 7NF, England or at [dac@ukc.ac.uk](mailto:dac@ukc.ac.uk).

Alan Ibbetson graduated in Biochemistry at the University of London in 1972, but quickly moved into computing. His masters degree is for research in message passing operating systems. He has been involved in UNIX support since 1980. Since 1990 he has managed a team of six people, including David, providing UNIX services to the campus at UKC. Reach him at [ali@ukc.ac.uk](mailto:ali@ukc.ac.uk).

Peter Collinson gained a Ph.D in Computer Science from the University of Essex in 1973. He then moved to the University of Kent as a Lecturer in Computer Science. He is responsible for making UNIX a major feature of computing activity on the Kent Campus. In 1980, UKC became one of the first sites in the UK to offer UNIX cycles to all computer users on campus. He left UKC in 1989 to start his own consultancy, Hillside Systems. He retains links with the University and is an Honorary Fellow of the Computing Laboratory. His email address is [pc@hillside.co.uk](mailto:pc@hillside.co.uk) and WWW page is <http://www.hillside.co.uk/>.

#### References

- [1] "Design and Implementation of the Sun Network File System", Sandberg et al., Proc. USENIX, Summer, 1985.
- [2] NetWare sales literature, Novell Inc.

- [3] LANmanager sales literature, MicroSoft Corporation.
- [4] "Scale and Performance in a Distributed File System", J.H. Howard et al., ACM Transactions on Computer Systems, Vol 6, No 1, February 1988.
- [5] "Scalable, Secure and Highly Available Distributed File Access", M. Satyanarayanan, IEEE Computer, Vol 23, No 5, May, 1990.
- [6] "Institutional File System Overview", T. Hanns, AIXTRA: The AIX Technical Review, pp 25-32, January, 1992.
- [7] "PC-NFS Administration Guide", 801-3977-10, March, 1993, SunSelect.
- [8] "Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD", G. W. Treese, Proc. USENIX, Winter 1988.
- [9] "Release notes for the TCP/IP Boot PROM Version 1.47", D. Koppen, EDV-Beratungs-GmbH, May, 1993.
- [10] "Bootstrap Protocol (BOOTP)", B. Croft and J. Gilmore, RFC 951, 1985.
- [11] "The TFTP Protocol", K. R. Sollins and N. Chiappa, RFC 783, 1981.
- [12] "The Athena Service Management System", M. A. Rosenstein et al., Proc. USENIX, Winter 1988.
- [13] "The Network Information Service", in System & Network Administration, SunOS 4.1 manual set, 800-3805-10, March 1990, Sun Microsystems.
- [14] "Domain Names - Concepts and Facilities", P. Mockapetris, RFC 1034, 1987.
- [15] "Kerberos Version 5 Revision 5.1", J. Kohl and B.C. Neuman, MIT Project Athena, Cambridge MA, September 1992.
- [16] "Foiling the Cracker: A Survey of, and Improvements to, Password Security", D. V. Klein, Proc. UKUUG Summer 1990 Conference, London.
- [17] "Message Send Protocol 2", R. Nelson and G. Arnold, RFC 1312, 1992.



# Monitoring Usage of Workstations with a Relational Database

Jon Finke – Rensselaer Polytechnic Institute

## ABSTRACT

The ability to monitor usage of groups of workstations is quite useful for planning growth, facility hours, staffing and other issues; but in our case, both the format of the data (*/var/adm/wtmp*) and the fact that the data was spread over hundreds of different workstations made any analysis difficult at best.

In this paper, we explore the use of a relational database to collect all the raw data, convert it to a standard form, and then provide selection tools to extract data sets. We also examine some ways to process session data to provide more meaningful reports and charts for administrators.

## Motivation

The primary campus computing system at RPI is a collection of over 400 color graphic workstations from both Sun and IBM, as well as some larger Sun and IBM *Timesharing* machines. The workstations are deployed in "workstation classrooms" of 25 to 30 machines, in smaller "dorm lab" clusters located in student housing, and as individual workstations on the desks of faculty and staff, as well as in laboratories.

The volume of data, on the order of one million records per semester, and the fact that it is spread over a large number of machines, makes it difficult to handle. In addition, we generally want to see usage patterns in a group of machines, and don't really care about the use of any individual machine in a group. We also have to deal with Suns and IBMs using different formats for their usage data (*/var/adm/wtmp*).

We wanted to be able look at the data in different ways. We need a way to determine what workstation clusters are filling up, and what sort of usage there is for any given time of day, or day of week. This will help our users determine when and where to go to find workstations, and assist us in figuring which buildings need more workstations and which ones need less! We are also able to compare if the users prefer one type of workstation over another, and if that holds for all sites.

Much of the funding for Rensselaer Computing System (RCS), was in support of computer enhanced learning, with a strong emphasis on teaching instead of research. A number of undergraduate courses are having their curricula revised to integrate use of the graphics workstations into the course. This increased interest in not only whether a workstation was in use, but the type of use, or at least the type of user who was using it. While the basic data contains a username, it does not have any demographic

categories of the user. Being able to find out more about the user is desirable.

## Solution

Several years ago, in response to a series of break-ins, we started a project to collect WTMP data in a central location to assist in locating connections from off campus, and odd usage patterns. This involved periodically "printing" the wtmp files to a virtual printer on our mainframe, where duplicate records would be discarded and new records would be saved for later analysis. While this got us through the immediate problem at hand, it did not take into account operational practices on the workstations (rolling wtmp files<sup>1</sup>), and sent huge amounts of duplicate data. This eventually overtaxed the print queuing system and jeopardized our print service (and the data collected was so difficult to work with), that we had to shut the data collection down.

A few years later, interest in gathering usage information had risen to the point where we needed to take another shot at the problem. While the previous solution (virtual printer) had been a failure, it did teach us some very valuable lessons, such as the need to handle the aging practices for */var/adm/wtmp*, and the need to only send back **NEW** data to the central collecting site. Given that I had just finished the Simon Hostmaster<sup>2</sup> project, collecting host usage data via Simon seemed to be natural.

<sup>1</sup>We periodically *roll* log files from say *wtmp* to *wtmp.0*. If there already is an old version (*wtmp.0*), we roll that to *wtmp.1*, and so on. Depending on the frequency of the roll, and the size and type of file, we keep from 2 to 9 old generations around.

<sup>2</sup>Simon Hostmaster is part of RPI's database driven system administration package, known as Simon, that manages all the hostname and address information for the name servers and host files.

Our solution breaks into three main areas. The first, data collection, deals with gathering all new data from each machines, doing some initial cleanup on it, and storing it into the database. The second area, data selection, deals with how we extract only the desired set of records from the database, provide additional preprocessing of the data if needed, and then pass it along for further processing. The third area, data modeling, is where we actually do some analysis on the data. This may involve building a virtual workstation lab, loading the usage data into the lab, and then analyzing the lab use.

### Collecting Data

The data collection is done using a program, *wtmp\_load*, that runs on each of the subject machines. It determines the last time we loaded data from the current machine, and then it scans through the wtmp file(s) for the first new record. If the first record in the wtmp file is newer than the last time we loaded data from the host, we back up one generation to the *wtmp.0* and check again. We keep backing up until we run out of old files, or find one that has older data in it. From that point, it starts reading forward until it finds a record that is later than the last time, and then loads the records into the database. If we had to back up to an older version of the *wtmp* file, we process each file in turn until we have loaded all the records.

A WTMP record is written for each signon, each sign-off, and depending on the actual operating system, for a number of other system events such as reboots, time changes, etc. It is important to note that there is a distinct sign-off record. We do not get session records in the wtmp file. Since later analysis will want to deal with sessions, we will attempt to build session records at collection time. By linking the start and end of a session at collection time, we don't have to do that work each time we analyze the data.

### Storing Data

All of the wtmp data collected is stored in the Oracle table **WTMP\_LOG**. We have defined the following columns<sup>3</sup>.

**username** *char(8)* The Username of the user.  
There are special usernames such as shutdown, reboot, etc....

**host\_id number** The Simon.Dns\_Domains. Domain\_Id of the host that these records are taken from.

**connect\_time** *date(7)* The time when the connection was made.

**disconnect\_time** *date(7)* The time when the connection was terminated. This value is usually only added via an update to an existing record.

<sup>3</sup>Some unused columns have been removed from this listing.

**line** *char(12)* The symbolic name of the device that the connection was made through.

**type** *char(1)* A flag indicating the type of connection.

**remote\_host** *char(16)* The name of the remote host involved with the connection. This may not be the full name due to truncation problems.

**remote\_host\_id number** The Simon.Dns\_Domains. Domain\_Id of the remote host, if it appears to be in Simon, (we have to make some assumptions here due to length limits.)

### Sign On Record Processing

When we process a signon record, we attempt to classify its session type (remote telnet, X console, remote X, ftp, etc) to simplify later analysis. This also helps eliminate operating system differences, which would complicate later analysis.

We also work to match up the partial remote host name from the wtmp record (esp relevant for the timeshare machines) with our own host database. Frequently the remote host name is truncated when it is stored, since at least some wtmp definitions only allow 16 bytes for the hostname. We declare it a match if there is at least one "." in the partial name, and if we can get an exact match with the first 16 characters of an RPI hostname. For names that match, we store the resulting host id in **Wtmp\_Log.Remote\_Host\_Id**. While we will miss hosts with very long names<sup>4</sup> and we may have some foreign hosts that match the first 16 characters of an RPI host, and so are miscounted, we should still end up with the **Remote\_Host\_Id** set correctly in most cases.

While we intended to run the *wtmp\_load* program on a frequent (actually, continuous) basis, runs were often weeks, sometimes months apart. This resulted in a lot of wtmp records to be processed, which in turn generated many queries to the Host-master tables to resolve partial names. Host name lookups go from right to left, first finding "edu", and then finding "rpi" and so on.

We finally added two levels of caching, first of the 16 character partial names, even if they did not match, and a level deeper, of individual parts of a domain name such as "edu" and "rpi". By seeding this lower level cache with "its"<sup>5</sup>, "rpi" and "edu", we cut the number of database queries in half. Both forms of caching made very noticeable improvements to the performance of the program, and reduced the load on the database server.

<sup>4</sup>There are 8 hosts at RPI that have a base name 15 characters or longer in a population of around 3700 hosts.

<sup>5</sup>Information Technology Services is the department that runs all of the RCS machines, so many of them are in the "ITS.RPI.EDU" domain.

### Sign Off Record Processing

When we process a sign off record, rather than insert another record into the database, we attempt to locate the corresponding signon record in **Wtmp\_Log** and set the **Disconnect\_Time** field. We only have the "line" information, so we have to look for all records for this host that have that "line" and **Disconnect\_Time** has not been set. While this works most of the time, we do on occasion encounter more than one record. This means that either the first session's sign off wtmp record never got written, or got lost somehow.

Missing sign off records are a source of error in the data. One way to reduce, but not eliminate, is to have the Signon processing first close out any pending records. It would also be good to mark all of these records as "suspect". Likewise with the sign off records, when you have more than one, the older ones should be marked as suspect, although if Signon processing closes open records, no extras should be found at sign off.

We encountered one type of system that never wrote a sign off record at all for a certain class of connection. Once we started working with the data, we quickly discovered this problem when it reported many people using the same X station at the same time. We also had problems with some FTP sessions not producing a sign off record.

### Other System Activity

We ignore everything except reboot records. When we process a reboot record, we close all pending session records for that host. Again, these records should be marked as suspect.

Between the attempts to resolve partial host names, and gaps in records caused by system crashes, the data being collected is by no means perfect, but for the most part, is clean enough to be useful. You wouldn't want to generate bills from it, but as long as you understand where errors can creep, in you should be ok. In attempting to track down some of these, we discovered an undocumented bug in some of the time conversion routines. Specifically setting the `tm_isdst` to -1 fixed this problem. Before that, it would intermittently add or subtract an hour.

### Selecting Data

Selecting data breaks into three parts: first determine which records we want based on attributes of the records; second determining which fields we actually want to return; last, any pre-process the records need to format columns outside of ways the database can handle, before passing the selected records on to the analysis section.

### Which records

Any of the columns in the **wtmp\_log** table can limit the selection of records. In fact, it is possible

to extend the selection choices by joining<sup>6</sup> the **wtmp\_log** table with other Simon tables. For example, via the **Username** column, we can join to the **Logins** table and only extract student users.

In practice, the first constraint is to just select the records for a single host, or a group of hosts. This is done by requiring that the **Wtmp\_Log.Host\_Id** is equal to a specific host id, or belongs to a specific host group. Host groups are described in more detail in a later section. In the case of an X station lab<sup>7</sup>, we instead select the records where the **Remote\_Host\_Id** belongs to the host group for the desired X station lab.

Another common constraint, are the starting and ending times. We actually select all records where the **Disconnect\_Time** is greater than the start time, and **Connect\_Time** is less than the ending time. This ensures that we get all records that fall into the desired time range.

Often, just specifying a host group and time range is enough. There are other cases where we want to place a fancy constraint, such as the join example from above, but more often we just want to look at a single type of connection. For example, when looking at the data for a workstation lab, you often only want to know if the console is in use. It doesn't really matter if a staff member is telneted to the workstation, as long as the console is available for use. In this case, we would add the additional constraint that the **Wtmp\_Log.Type** would be "X", indicating an X console session.

### Which Columns?

Given that we have determined which records we are going to select, the next step is to determine which columns to select. We always want the session start and end times; the rest is up to the question we are asking at the time. In the current implementation, we return a linked list of a structure that has three different columns, as well as the start and end time for each record. By convention, the first column is a numeric, and the other two are strings.

A common choice for a workstation lab, is **host\_id**, **Username** and **User\_Class**. In point of fact, we don't do anything with the **Username**, and due to privacy concerns, we could not publish a report with usernames in it. **Username**<sup>8</sup> is actually used in the database itself to find out the

<sup>6</sup>With a relational database, you can "join" the contents of one table with the contents of a second table based on a common column between the tables. This is a very powerful function.

<sup>7</sup>For X station labs, the wtmp records are collected on the actual machine providing the CPU cycles, and the X station is considered the remote host in this case.

<sup>8</sup>We manage all user accounts via another Simon module, based on data from the Registrar. This enables us to match up a given username with the corresponding student records, which has been valuable cases as well.

classification of the user (Freshman, ..., Phd, Fac/Staff). It is in this type of join that the power of the relational database comes into play. This has enabled us to study for example, whether lab users live off campus, on campus or on campus in a dorm with a workstation lab. The potential here is amazing. For the sake of example, assume we have a table **User\_Info** with the following columns defined:

**Username** *char(8)* The Unix Username. There is a record here for all active user accounts.

**Classification** *char(4)* The current classification (FR,SO,JR,SR,Grad,Misc) of the userid. The status "Misc" includes faculty, staff and guests.

To get the classification, we would select something like the following:

```
Select Host_Id, Classification,
       Connect_Time, Disconnect_Time
from Wtmp_Log, User_Info
where Host_Id in (Sub Selection)
   and Connect_Time < $ENDTIME
   and Disconnect_Time > $STARTTIME
   and Type='X'
   and Wtmp_Log.Username=
                               User_Info.Username
order by Host_Id, Connect_Time
```

The statement (*Sub Selection*) actually refers to a nested select statement which returns the list of

host\_ids for the group we are interested in. This will be discussed in detail in the section on host groups. The first two "and" statements establish the starting and ending time constraints, and the third "and" statement sets the type constraint. In the last "and" statement, we joined the **Wtmp\_Log** table with the **User\_Info** table to get the **Classification** returned with each of the records.

### Fixing Data

While we can do a lot with joins in the database to get what we want, sometimes, there are ways of classifying data that seem too complicated to get directly from the database. For this, we simply run the data from the selection process through a routine that converts one of the data fields in place to some new classification.

An example of this, is when we wanted to look at what sort of people were using our remote access (timesharing) Unix service. We convert **Remote\_Host\_Id** into one of the following cases: Terminal Server, On Campus RCS Host, On Campus non RCS host, Student machine or off campus. With a combination of host groups, string compares, and other smoke and mirrors, we were able to convert the remote host info into what we wanted, and the existing analysis routines were happy to produce results for us.

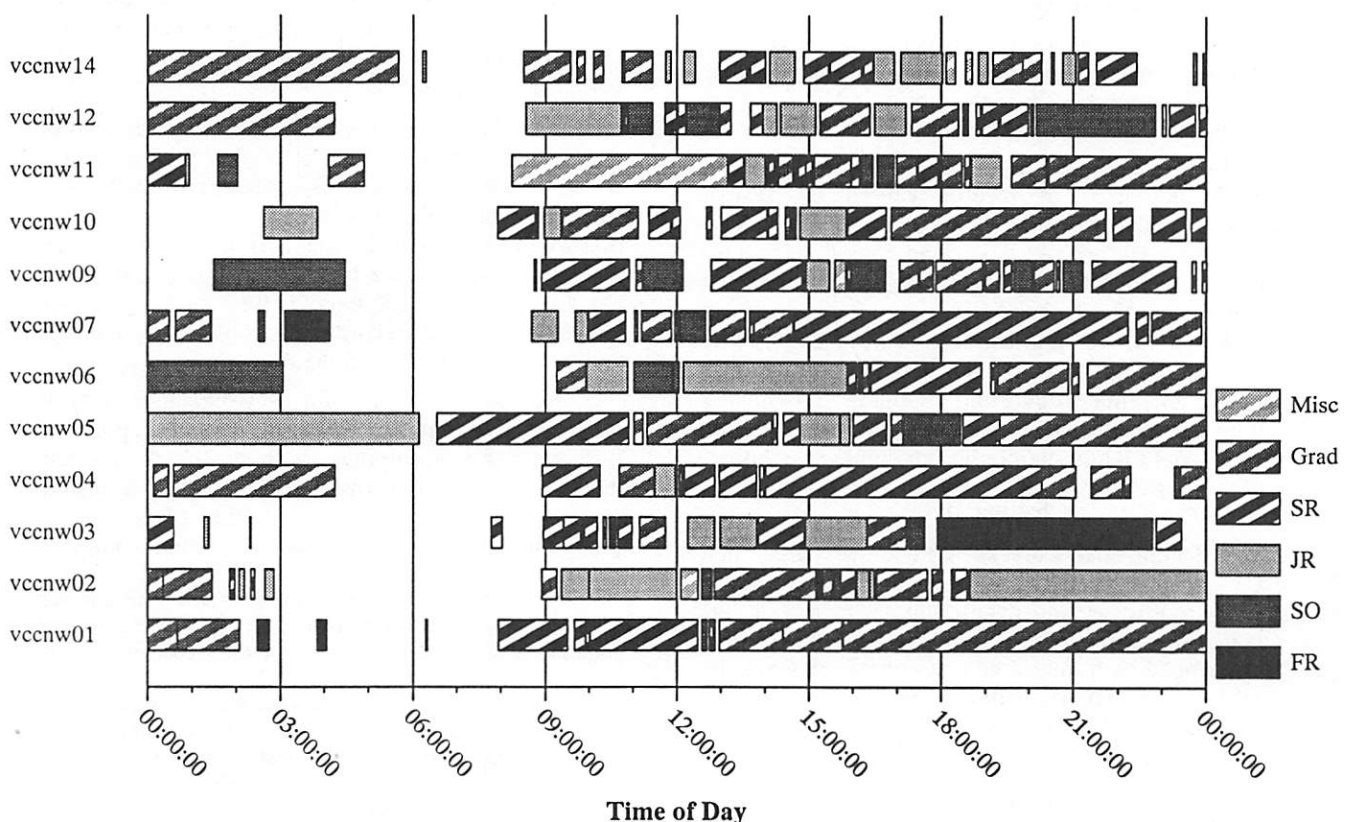


Figure 1: Raw Session Data

Sometimes, at this stage, we simply dump the records we have selected into a file to allow for analysis with other products such as SAS.

### Verifying the Data

Now that we have selected a set of rows, and figured out which columns from those rows are of interest, we wanted some quick way to look at the lab use before we actually start modeling it. To this end, we generate a bar graph like Figure 1. In this case, we take a small workstation cluster, and look at all the records for a particular day. We select the `host_id` and a user classification, which is derived from the username. Each of the hosts has a set of bars that correspond to a user session. The shading and patterning indicates how we classify the user (in this case.) The time axis runs from midnight to midnight. This type of output has proven very useful in finding problems with the raw data. The X station problem mentioned earlier, showed up as more and more sessions overlapping. Since that is not possible, it indicated a problem with the data. This also can show unexpected gaps in the data. This is often due to a broken workstation. While this was not intended for the formal reports, this format has proven useful to show the user mix in the labs. The actual output is much more impressive in color.

### Modeling Data

One of the initial objectives, was to generate a graph showing number of machines in use, at any given time of the day or night. Logically, if we take the graph in figure 1, and draw a vertical line at some particular point in time, then count the number of times it intersects a horizontal bar, we then have a user count for that time. We advance the vertical line to a new point, some fixed distance from the previous point, and repeat. This isn't a new concept; I seem to recall something like this from a freshman calculus class, long ago.

Moving that model into the computer is mostly a matter of picking some data structures. For ease of processing, I broke up the time line up into a set of discrete "buckets" with an array element for each bucket. Given that we had the start time, end time and bucket size (or duration), it is trivial to figure the number of buckets or elements in the array. For each record, we simply converted the start time to an array index, and looped through until we hit the end time. In practice, I ran several of these array structures, a master array (all hosts), a linked list working on the primary key (such as the `host_id`), and a second linked list working on the secondary key (such as the classification).

If we take the master array, and use 5 minute buckets, we get a simple graph like Figure 2. If you compare this with the data shown in Figure 1, we can see where every machine is empty at about 6:20

AM, and then the lab is in constant use for the rest of the day. There is a slight dip at lunchtime, which is a little more visible on Figure 2, than it is on Figure 1.

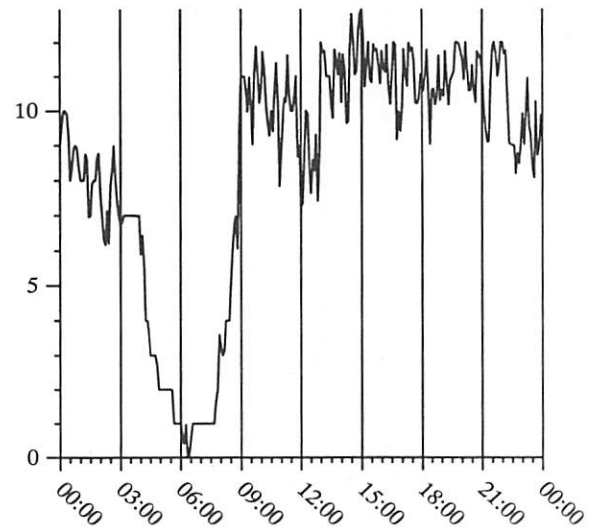


Figure 2: Simple Lab Use

We can also dump more than one of the chains on the same axis. Putting the 13 different hosts on one axis, where they only value they can have is 1 or 0, would be pretty boring, but if we take the other chain, classification, we display (see Figure 3) undergrads as a solid line, grad students as a dotted line, and fac/staff as a dashed line.

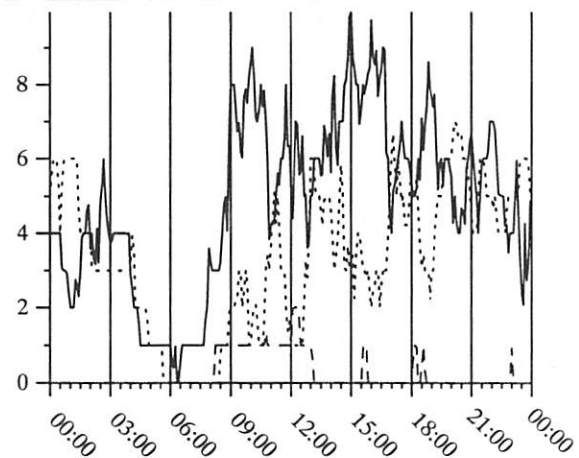


Figure 3: Use by Category

### Post Processing the Model

Well, all of this provides a start, but what you often want to do is look at the average use of a site, so the program can take data for say 5 days, and calculate a mean value for each bucket, and since Rensselaer is an engineering school, figure a standard deviation of the mean, and put that on as well, as seen in Figure 4.

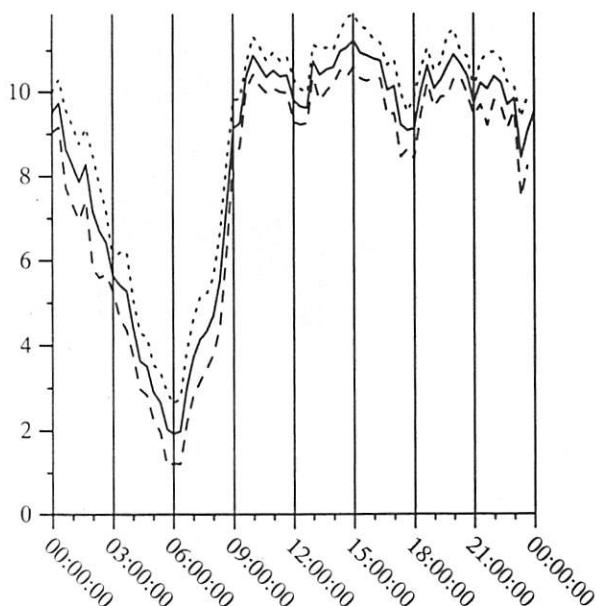


Figure 4: Mean Usage

You can also do things like put two different labs on the same axis to compare usage patterns, overlay different days of the week to look for scheduling differences, etc.

We found the ability to overlay two different labs on to the same axis quite useful. One of the objectives of RCS, is to allow the student to work on any of several different platforms (Sun, RS/6000, X station) and move between them on a daily basis. This allows the students to "vote with their feet", as to what is the preferred workstation. Given a choice between an X station and a workstation with local display, there is appears to be a preference for the

workstation, except in the cases of dorm labs, where convenience appears to win out over technical attributes. An example of this, is in Figure 5, where we take a five day average of workstation console use (the solid line), and then on the same axis, put a five day average of X station (Remote host) use (the dashed line).

### Implementation

The wttmp logging project got put on hold this past winter, when the database machine ran out of disk space. Since a new disk and a new database machine were imminent, we stopped collecting data (and letting it accumulate on each individual machine) until we could move to the new machine. That move is currently scheduled for mid August.<sup>9</sup> At that point we intended to start with a clean slate and start collecting data from all machines, all of the time. Before we ran out of space, we collected over 1,000,000 session records from over 400 machines. One critical item for performance, is an index on **Host\_Id** and **Line** since this is the most common query for *wttmp\_load*.

The data collection will be done with the *load\_wtmp* program. I expect that when operation resumes, we will run it in "sleep" mode. In this mode, when it first runs, it will connect to the database, update whatever records are available, close the database connection, and sleep for some length of time. It will periodically wake up to see if anything has changed in */var/adm/wtmp*, and if so, process any new records, and go back to sleep. It also

<sup>9</sup>I think we need to work on our definition of *imminent*

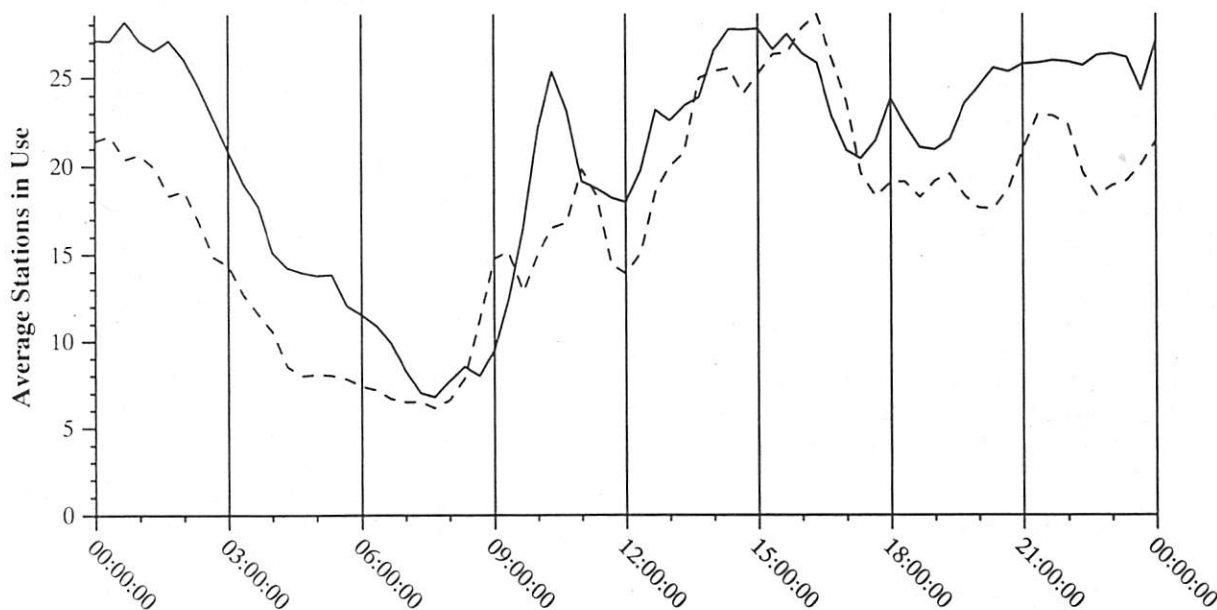


Figure 5: Workstations vs X Stations

has to handle the *wtmp* file be rolled out from under it. This should allow us to keep the database up to date, and this may also provide for a periodic health check for machines.

One of the parameters we will be working with, is the length of time the process "sleeps" on different types of machines. For a single workstation, the average session is 25 minutes or so, so a frequent check (every 5 minutes) would let us keep up to date, but not put too much load on the database machine. Things are different on the remote access machines. These run on the order of 50 sessions an hour, so a frequent wake up, would result in a lot of database activity. The other approach is to set a long interval (such as 24 hours) between updates. We are assuming that there is non trivial overhead in establishing a database connection, so that there are gains in batching records. One thing we need to avoid, is a bunch of machines all attempting to dump data at the same time. Since they are contending for some of the same resources on the database machine, this is likely to cause performance problems. One performance win we do get with sleep mode, especially on the remote access machines, is that they are building up a cache of host names over time.

The other half of this project is the *lab\_use* program, which is used to make the queries and generate the graphs. This program is a product of evolution, once we produced one set of reports, the vice provost would ask another set of questions, which would involve more changes. As time went on, we learned what types of queries made sense with what types of output displays. For instance, the graph of mean use doesn't really make sense for just a few days, nor does a simple graph such as Figure 2 make any sense for three month time period. We did learn some things in the process, and while we made some mistakes, we also did some things well.

### What We Did Wrong

In general, the single biggest problem with *lab\_use*, is that we did not have a clear design specification in mind. As a result, the program has had some major internal restructuring, and some cruft from earlier versions still lingers. In some respects, this was unavoidable, as this was a crash project to get some numbers to the administration.

As we started writing *lab\_use*, we got tired of providing a huge number of command line parameters, so we eventually built a structure to hold common queries, so we could just specify an entry from the struct. While quite useful in the beginning, this has turned out to be difficult to update and now serves to slow development in some areas.

Another problem, stemming in part from the development approach we took with *lab\_use*, is the original lack of program structure. This has required us to go back and split up the program into different

modules, to allow better reuse of code. As a result, not all of the graphing routines use the same color selection routines, some processing routines make graphing calls directly, while other simply fill an array and pass that off to be plotted and so on. However, given what we have learned, we could now go back and redesign it.

### What we did right

On the other hand, a number of things have worked out well. For instance, the host group and time constraints have worked quite well, and certainly should be kept in any revision. Another useful development is a set of time conversion routines which convert the time values we get from oracle into a handy internal format<sup>10</sup> and back again. While these are pretty simple routines that make heavy use of the existing C routines, they do handle some vendor differences, assist in default formats, allocate space as needed and so on.

Another big win on this project, is the use of *jgraph*[1], written by Jim Plank at Princeton has proven quite useful and quite a time-saver. If you need to generate graphs from a program, consider this package. This also made it easy to scale and edit the figures for this paper.

As the program evolved, some structure did start to appear: libraries to perform calculations and store the results into standard structures; libraries to convert the standard structures into *jgraph* input files. The time library mentioned above and one to help with host groups. One of the snags we hit was in labeling all of the graphs. Since there are 4 or 5 different output formats, and dozens of different selections defined, we started to store graph labels in an oracle table, so you can specify the format and the selection, and find the appropriate label for that graph. I expect that this approach would work well for storing the queries.

One of the biggest wins was basing the whole project on the relational database. Besides the options that joining other tables provide, it gives us a lot of data independence. While many of the internal variable names reflect the original selection choices, that is hidden from the end user so we can process other columns than the ones in *wtmp\_log*.

### Future Directions

This is the type of project that will never end. This fall, I expect to start the data collection on all machines, into the newer, bigger database machine. The *lab\_use* program is likely to remain unchanged until the next round of questions start coming from the directors. At that point I hope to continue some of the cleanup and restructuring. Given the existing queries that it supports, I want to identify the

<sup>10</sup>What could be more convenient than the number of seconds since 1970, stored in a long integer...

attributes of each type of selection and displays, and come up with a more idiot proof way of determining output options.

Another are I want to work on, is the driving force of the original project, rapid investigation of violations of conditions of use. Right now, we are sometimes faced with questions like "when and where has this user signed on". Given the hundreds of machines, this is non trivial to go out and collect; with the database, this will be trivial. There are other related questions that come up, and while I would rather not deal with them, we don't seem to have much choice.

One popular feature of our old mainframe system, was the *TermIdle* screen. This enabled people to see which terminal rooms had available machines. This was possible since every terminal was hardwired directly to the mainframe. It has been a long time since we have been able to provide this, but this project may provide the means to do it. This would require the *wtmp\_load* running in sleep mode with a short check interval. In addition to the **WTMP\_LOG** table, it could also update a much smaller table, with one entry per host with the current state of the machine. We might also want to keep the time of the last successful session, as a complaint of the old *TermIdle* system, is that during crunch times, it gave the number of broken terminals in each room.

Another area to explore, is using these tools to track data other than Unix *wtmp* files. Any session based logs, such as the ones from our terminal servers, or resource pool logs from the campus phone switch could be able to be loaded into Oracle, and at that point the existing tools should be able to do all the same analysis on these records, as it can do on *wtmp* records.

### Related Projects

For the past four years, RPI has been developing a suite of tools called Simon, to assist in the management of UNIX systems. Some parts of the Simon project were used in this project. Those, and other additional things are described below.

#### Host Name Database

The Simon Hostmaster[2] project is a set of programs and oracle tables that assists in the maintenance and generation of the resource record files for named, and host table files. For this project, we are just interested in the host naming part which is done with the **Dns\_Domains** table. The column of interest here are:

**domain\_id number** A unique identifier for the given domain/node. This identifies this particular entry on the DNS tree. This will be drawn from the *simon.peoplecount* sequence

**name char(64)** The simple text for this node name. It is an unqualified string with no "."'s in it.

**parent\_id number** The domain\_id of the parent to this node. Given a node, you can build a name backwards by searching for the parents. Alternately, given a node, you can find all children.

Each node has a parent, with the root node having a domain id of 0. Consider the **Dns\_Domains** records in Figure 6.

Name	Domain Id	Parent Id
edu	245	0
rpi	246	245
its	302	246
cs	442	246
jon	752	302

Figure 6: Dns\_Domain Table Excerpt

The host *jon.its.rpi.edu* has the **Domain\_Id** of 752. Given a host id (*domain\_id*), we can run links backwards to build up the fully qualified hostname. This structure allows us to have as many domains as we want, and go as deep as we need to.

#### Host Groups

The host groups proved very useful in selecting hosts. They are described in more detail in another LISA paper[3].

#### Jgraph

For the graphical output, the *jgraph*[1] program written by Jim Plank at Princeton has proven quite useful and quite a time saver. If you need to generate graphs from a program, consider this package.

#### Availability

All the source code for the printmaster suite of programs, as well as table definitions, source code and additional reference material for the Hostmaster and host group modules are available for anonymous FTP from *ftp.rpi.edu*. See the file */pub/its-release/Simon.Info* for details on where to find everything. Some papers and presentations that discuss other parts of the Simon project are available in */pub/its-papers*.

If you have AFS available, you can browse many parts of the Simon tree. Look in */afs/rpi.edu/campus/rpi/simon/logging* for this project, and */afs/rpi.edu/campus/rpi/{sql,sandbox,netjack}* for other related parts. The anonymous ftp tree is also available in */afs/rpi.edu/campus/rpi/anon-ftp/1.0/common*.

If you just want to poke around, some information is available via

<http://www.rpi.edu/~finkej/Simon.html>

#### Author Information

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming with a BS-ECSE. He continued as a full time staff member in the

computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past 3 years. Reach him via US-Mail at RPI; VCC 315; 1108th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu.

#### References

- [1] Plank, James S. "Jgraph - A Filter for Plotting Graphs in PostScript" *Proc Winter 93 Usenix*, 1993.
- [2] Finke, J. "Simon System Management: Hostmaster and Beyond" *Proc. Community Workshop '93*, Simon Fraser University, June, 1993.
- [3] Finke, J. "Automating Printing Configuration", *Proc., USENIX LISA VIII* 1994.



# Adventures in the Evolution of a High-Bandwidth Network for Central Servers<sup>†</sup>

*Karl L. Swartz, Les Cottrell, and Marty Dart – Stanford Linear Accelerator Center*

## ABSTRACT

In a small network, clients and servers may all be connected to a single Ethernet without significant performance concerns. As the number of clients on a network grows, the necessity of splitting the network into multiple sub-networks, each with a manageable number of clients, becomes clear.

Less obvious is what to do with the servers. Group file servers on subnets and multi-homed servers offer only partial solutions – many other types of servers do not lend themselves to a decentralized model, and tend to collect on another, well-connected but overloaded Ethernet. The higher speed of FDDI seems to offer an easy solution, but in practice both expense and interoperability problems render FDDI a poor choice. Ethernet switches appear to permit cheaper and more reliable networking to the servers while providing an aggregate network bandwidth greater than a simple Ethernet.

This paper studies the evolution of the server networks at SLAC. Difficulties encountered in the deployment of FDDI are described, as are the tools and techniques used to characterize the traffic patterns on the server network. Performance of Ethernet, FDDI, and switched Ethernet networks is analyzed, as are reliability and maintainability issues for these alternatives. The motivations for re-designing the SLAC general server network to use a switched Ethernet instead of FDDI are described, as are the reasons for choosing FDDI for the farm and firewall networks at SLAC. Guidelines are developed which may help in making this choice for other networks.

## Introduction

In a small network, clients and servers may all be connected to a single Ethernet. This simple approach provides fast and relatively reliable communications between clients and servers. Unfortunately, it does not scale well – performance may suffer as the addition of more hosts (and thus more traffic) brings on network congestion, reliability may suffer as the result of there being more pieces in the Ethernet that could fail in a manner which impacts the entire network, or simple physical limitations may be reached. Splitting the network into multiple subnets works for the clients, but what to do with the servers may be far from obvious.

## Keeping Servers Close To Clients

Ideally, one wishes to preserve the simplicity inherent in having clients reach their servers over a single Ethernet. Forcing traffic to traverse multiple networks adds delay and introduces additional opportunities for failure, especially if the routers become congested. NFS, in particular, is very sensitive to congested intermediate routers and responds to the situation most ungracefully [1,2].

Many vendors would like everyone simply to buy workgroup servers and distribute them amongst the client networks. This can work quite nicely if groups within your organization are neatly compartmentalized and you can afford to buy servers for each of them, but substantial interactivity between groups will reduce the effectiveness of this solution. Institutional databases take this to the extreme, yet they are commonplace.

## Inherently Centralized Services

SLAC is an experimental physics laboratory, and the physics data is at the core of our computing. Today that means a few terabytes of data in four tape silos, with a new experiment and hundreds of terabytes of data looming ominously on the horizon. With various groups within the laboratory working together to collect and study such large amounts of data from a single experiment, departmental servers are not viable. A similar situation exists in many other organizations – airlines and their reservation databases are a striking example, though most any organization probably has examples.

Supercomputers pose a similar problem; the only difference is a change in perspective, with computing cycles instead of data as the shared, central resource. More common are mainframes, which represent a mix of shared data and computing cycles.

<sup>†</sup>This work supported by the United States Department of Energy under contract number DE-AC03-76SF00515, and simultaneously published as SLAC-PUB-6567.

Even in a more enlightened world, where such dinosaurs have been banished to Hollywood, centralized services will persist. Firewall gateways to the Internet and NetNews (which really is just another big database) come to mind, as do mail routers, even if departmental mail servers handle part of the load. The problem isn't likely to go away.

### Multi-homed Servers

Connecting a few large servers to multiple networks – multi-homing them – appears to provide a reasonable compromise. Auspex servers are designed with this in mind, and Sun, for example, seems to encourage using their larger servers this way. SLAC has implemented multi-homing on a limited basis<sup>[1]</sup>, but, like any compromise, this solution is not perfect. The added complexity is perhaps the worst problem – even after investing a great deal of effort, multi-homing causes confusion amongst users and administrators, while certain applications simply don't work on multi-homed hosts.

Availability of I/O slots in the servers and the cost of additional Ethernet interfaces places further constraints on widespread multi-homing of servers. While connecting a few big servers to a few busy networks helped, a lot of smaller servers talking to a lot of quieter networks still created a tremendous load. Individual server-network pairs could not justify additional direct connections, but in aggregate, the server network was still very congested.

### A Bigger Pipe

The need for higher bandwidth amongst the central servers and core routers suggested a switch to something faster than Ethernet. Conventional wisdom suggested a switch to FDDI<sup>[3]</sup>, with bandwidth at least an order of magnitude greater than Ethernet. (100 Mb Ethernet hadn't entered the scene yet.) Interfaces were expensive and availability spotty, but there seemed to be a strong movement towards FDDI and we felt the situation would improve by the time we needed a substantial FDDI investment.

Building an FDDI network for the central servers would of course mean there would be at least one router hop between the servers and the Ethernet-based clients. In part, we hoped to minimize the risk by giving every Ethernet a direct connection to the FDDI ring, keeping client-server communications to at most one router hop, and by using fast routers<sup>1</sup> that should easily be able to keep up with an FDDI and a handful of Ethernets. For the common 8 kB NFS reads, the greater maximum transfer unit (MTU) of FDDI would also mean the router would only see two packet fragments instead of six, reducing the vulnerability of a packet to fragment loss.

<sup>1</sup>Each FDDI router at SLAC is a Cisco AGS+.

Various features of FDDI promised further reliability benefits. The ability to "heal" the ring by wrapping back upon encountering a failed node was particularly attractive, as was the ability to further cordon off problems by isolating servers behind wiring concentrators.<sup>2</sup>

Overall, we felt that FDDI represented an improvement in reliability despite the added router hop. We still had the option of directly connecting servers (i.e. multi-homing them) to networks for which highly reliable connectivity was paramount.

### Experience with FDDI

After several years, our experience with FDDI has been less idyllic than we had hoped for. Prices have come down somewhat, but FDDI interfaces and other devices are still expensive. Implementations have been buggy and have exhibited various interoperability problems. Identifying and solving problems has been hampered by inadequate diagnostic and monitoring tools as well as the ignorance of vendors and ourselves. When the networking is working well, it's not uncommon to find that other software is not prepared to take advantage of the faster speeds.

We began with a ring composed of three devices, shown in Figure 1: a Cisco AGS+ router, a Sun SPARCserver-390 with a Sun FDDI/DX interface, and a DEC wiring concentrator. A VAX-9000/410 (running VMS and Multinet) was connected via the concentrator. Despite the mixture of vendors, things worked pretty well, which was a good thing since we had no way to diagnose problems.

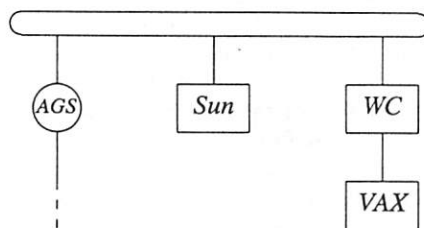


Figure 1: Initial SLAC FDDI network

The next step was to add a SPARCserver-2 and an RS/6000-340. Unlike Sun's VME bus FDDI/DX interface, the SBus FDDI/S card only implements a single attached station (SAS), so it had to be connected to the DEC wiring concentrator. The RS/6000 had IBM's optional second board which allowed a dual attached (DAS) connection, i.e., directly to the ring, but we had decided that we only wanted routers and concentrators on the ring, so it was connected to the concentrator – the FDDI

<sup>2</sup>FDDI offers a variety of redundancy and fault isolation features. See [3] for a good, introductory discussion of FDDI's features.

specification allows DAS devices to be connected in a SAS configuration. (We later connected the SPARC 390 this way too.)

### Big Blue Blues

The RS/6000 is where we encountered our first trouble. Always eager to annoy, AIX came with FDDI support, but it didn't work – a special micro-code update was necessary. We obtained that and installed it, but the FDDI still didn't work. While checking the cabling, it was noted that the fiber was plugged into the second card of the IBM adapter. This adapter is composed of one MicroChannel card which implements the bus connections and a SAS interface, while a second, optional, MicroChannel card adds the DAS capability. Despite claims that it shouldn't matter, it was found that moving the fiber to the main adapter card allowed the connection to work. This problem was to become quite familiar, not only as technicians miscabled RS/6000s (about half the time), but also in some more perplexing ways, to which we'll return.

A channel connection between an RS/6000 and our mainframe, an IBM ES/9000 running VM/CMS, was the next source of trouble. While not really an FDDI problem, it was a direct result of FDDI, and dramatically illustrates the costs of FDDI. Specifically, the VM system is still a critical part of SLAC's computing environment, and a prime candidate for an FDDI connection. Unfortunately, IBM wanted \$80,000 for a 370 FDDI adapter! We figured we could build one with an RS/6000, routing packets between FDDI and the mainframe channel for well under half that price. We could, and did, but it's always been cantankerous.

While the FDDI support in AIX still exhibited a number of problems, they were minor enough that we felt we could work around them until fixes arrived from Austin. Things seemed stable enough for us to put our main NFS fileserver, an RS/6000-970, onto the FDDI ring, and later our two Oracle servers. Despite the dramatically higher load, things seemed to be going well – for a while. Then, we started noticing occasional NFS hangs. They seemed to occur during periods of high load, and to last for about 90 seconds. Eventually, we managed to correlate them to an FDDI adapter error on the RS/6000-970, and found examples of the error in the logs of other RS/6000s. IBM didn't seem to know what was going on, but they did proffer a blizzard of new system patches.

All of our RS/6000s with FDDI had been ordered with the second card to provide DAS capability, and all of them were connected SAS-fashion via a concentrator. Once again, that extraneous card came to mind as a possible culprit. A check revealed that most of the RS/6000s were miscabled – but now they worked, at least most of the time! We had switched to a Cabletron wiring concentrator

somewhere along the way, and apparently it wasn't as fussy about cabling as the DEC concentrator had been. Still, that extra card was a suspect, so we called in IBM field service to have the board removed from the NFS server. Upon arriving, the technician refused to remove the board, claiming the extra board provided additional reliability. He'd obviously heard about FDDI's ability to wrap around failed devices, which only works for dual-attached stations, but he didn't understand enough to appreciate IBM's own recommendation that hosts not be directly attached[4].

We eventually removed the extra card from one machine, but the problem persisted. An upgrade to AIX 3.2.5 appeared to produce a reliable FDDI connection for our RS/6000s at last, but not without one last round with the extra cards – while nobody could prove any problem with SAS-cabled DAS systems running older versions of AIX, IBM Austin knew for a fact that this configuration would not work with AIX 3.2.5. We finally removed all of them once and for all and installed 3.2.5. Only one bug remained, and, while confusing, it was harmless if you were aware of it. After a mere 18 months, we had what seemed to be production-quality networking.

At least that's what we thought. Then we suffered another 90 second outage of the file server, again correlated to an FDDI adapter error but not traceable to any other event on the network. While the problem is much less obtrusive now than it was previously, due to the lower frequency of occurrence, it remains unresolved.

### Sun Brings Darkness To The Fiber

IBM wasn't the only vendor to bring grief to our FDDI effort. A new SPARCserver-10, again with Sun's FDDI/S adapter, was installed, its FDDI interface was configured, and all was well – for a minute or two. Then the wiring concentrator started having convulsions, and the entire ring crashed. Disabling the Sun's FDDI interface restored the ring, though at least once a wiring concentrator crashed hard enough to require cycling its power. Replacing the FDDI/S adapter was ineffective. A patch was obtained from Sun which addressed a problem with frequent ring state transitions on the FDDI/S adapter, but the only effect seemed to be to reduce error messages on the console. (We later discovered that that is all the patch was intended to do!) The SPARC-2 had never had these problems, but for other reasons was no longer on the FDDI, and a lot had changed since it had been. Puzzled, and needing to get other work done, we temporarily shelved the problem.

When we came back to it a few months later, we started from scratch. SunOS was re-loaded from CD ROM and the FDDI/S software was installed. Every SPARC 10 and FDDI patch we could find was applied. We scheduled an outage, checked and re-checked all the cables, and finally enabled the FDDI

interface once again. It worked. The network map stayed green, the concentrator hummed along peacefully, and everything did exactly what it was supposed to do, even after a week had gone by.

In reviewing what had changed in the past few months, we found that our Cisco routers (there were several on the ring by now) had received a microcode update that fixed a hyper-sensitivity to ring state transitions – exactly the situation which that first Sun patch was supposed to have addressed. We subsequently found a review of SBus FDDI adapters which documented the problem of frequent ring resets with the Sun FDDI/S adapter[5]. It appears that the Sun adapter bug had been aggravating the microcode bug in the Cisco routers, which then not only crashed themselves but also took out the Cabletron wiring concentrators – something which isn't supposed to happen.

### What Went Wrong?!

In spite of work dating back an entire decade, FDDI clearly is not mature yet. High costs have undoubtedly inhibited widespread deployment of FDDI, and our mixture of products from at least half-a-dozen vendors is probably a greater interoperability challenge than we would have liked. Considering the number of substantial interoperability problems with Ethernet, even after more than two decades of use in far more diverse environments[6,7], it shouldn't come as much of a surprise that the more esoteric (and far more complex) FDDI still has a lot of bugs to be uncovered.

### Monitoring and Troubleshooting FDDI

Debugging FDDI problems and monitoring the health of the network has also proven to be problematic due to a lack of experience, compounded by inadequate tools. The case of the IBM technician who knew only the sparsest details of FDDI is by no means an isolated case. The SPARC-10 case demonstrated that sufficiently in-depth experience with FDDI within SLAC was equally lacking.

The acquisition of a Tekelek FDDI analyzer helped with debugging to some degree, and with testing new equipment. It's mainly oriented towards the hardware, though, and the lack of a device which understands the higher-level protocols has been a handicap. (Network General's Sniffer now has an FDDI option which brings this capability to FDDI.)

Routine monitoring is also a problem. For Ethernets, we put an NAT Ethermeter on each major segment and use RMON to collect a variety of performance and error information. Values which exceed certain thresholds, as shown in Figure 2, trigger alerts. Further data is collected via SNMP from bridges and routers and from interesting hosts[8,9]. Alas, no FDDI "Ethermeter" is available yet, and FDDI MIBs in the various devices are

incomplete or non-existent. Even if we did have the data, the lack of baseline information makes problem threshold determination difficult, as compared to Ethernet, which by now is well understood.

value	alert if exceeds
CRC and alignment errors	1 in 10k packets
total utilization on a network	10% for the day
broadcast rate	300 per second
(shorts+collisions)/good_packets	10%
packet losses from ping tests	1% in a day

#### 2a. Ethermeters (RMON data)

value	alert if exceeds
CRC and alignment errors	1 in 10k packets
buffer, controller overflows	0

#### 2b. bridges

value	alert if exceeds
total interface input errors	1 in 10k packets
collision rates	10% of packets
CRC and alignment errors	1 in 10k packets
buffer, controller overflows	0
in/out queue drops and discards	0
ignored packets	0
interface ping packet losses	1%

#### 2c. routers

Figure 2: SNMP data and alert thresholds

### FDDI Performance

Given a functional FDDI network, another hurdle is getting software to take advantage of it. Some kernel tuning was required, especially on AIX, to allocate enough mbufs for the higher data rates, to increase the default TCP buffer size, etc.

Using the larger MTU is also important – one experiment was using NFS to read data from the VM system to the VAX 9000 (admittedly not an ideal choice) and getting horrible performance while making prodigious use of CPU cycles on the mainframe. It was found that, even with a direct FDDI link, the Multinet NFS software on VMS was using a 512 byte read size. Forcing a 4096 byte size, which fits nicely in FDDI's 4352 byte MTU, improved the performance dramatically. AFS, which SLAC is starting to deploy, similarly used a small MTU, though not so small as to be inefficient even for Ethernet. In this case, the MTU was not tunable but the problem was fixed in AFS 3.3[10,11].

Router performance with FDDI was disappointing as well, at loads well below what Cisco's numbers suggested should be easy for an AGS+ to handle. While our findings weren't rigorously documented, one subsequently published test demonstrated only 12 Mb/sec when routing from one FDDI to four Ethernets[5]. An average of 3 Mb/sec per

Ethernet is 30% of the bandwidth, not a light load, but still below what one could expect from an uncongested Ethernet[12].

Investigation of the router statistics turned up a large number of dropped packets on the FDDI inputs. It turns out that Cisco routers allocate the same amount of buffer memory for each interface, regardless of the bandwidth of the interface[13]. Thus, while the Ethernets had an abundance of buffer space, the FDDI was starved, and incoming packets were being dropped, which, as mentioned in the discussion of avoiding router hops, is particularly bad for NFS traffic. Despite tight budgets, we were forced to upgrade our existing routers and acquire additional routers, which eased the problems.

### Analyzing the Server Network

The ongoing problems and disappointing performance of FDDI, along with the cost of equipping our growing number of servers with FDDI interfaces and adding concentrator ports for them, led us to revisit the decision to use FDDI for our server network. Because of the large volumes of data moving across our network, and because UNIX represents only a small (but growing) part of the computing environment at SLAC, our performance monitoring and problem detection efforts have tended to focus on the networks[8, 9] rather than individual servers.<sup>3</sup>

The most interesting information for studying the server network proved to be the "Top 10 Talkers" report, which, for a given Ethernet segment, shows the top ten source/destination pairs seen in packets during a given hour, with summary reports for yesterday and for today so far. Most pairs on the server network tended to involve at least one router – the information is based on hardware (MAC) addresses, not IP addresses, so traffic going on or off the network has a router on one end.

A significant percentage of traffic involved two routers, with one being the firewall router which connects the SLAC network to the Internet. One could view this as transit traffic which shouldn't be on the server network. A more general view is to look at the Internet as being just another service, albeit a rather special one, with the firewall router being the server providing that service.

Notably, no pair dominated the traffic on the network, and only one pair (the firewall router to our best connected internal router) consistently exceeded 10% of the total traffic. Except for a few short-term anomalies, intra-server traffic only occasionally made the top 10 list, with the RS/6000 fileserver being involved in most such cases.

<sup>3</sup>An interesting approach to automated system monitoring (which could be applied to network performance monitoring as well) is contained in [14]. For a discussion of NFS performance monitoring, see [15].

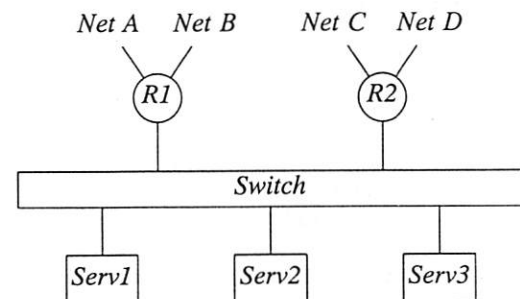
Further analysis was done using Sun's `etherfind` utility and by examining the usage count field in `netstat -r` output on major servers. This further bolstered the model of lots of servers, each contributing a modest (in terms of Ethernet bandwidth) amount of traffic to the server network, aimed at a variety of clients.

This finding suggested that providing bandwidth greater than that of an Ethernet to each server was unnecessary. Aggregate bandwidth of the server network was what we needed.

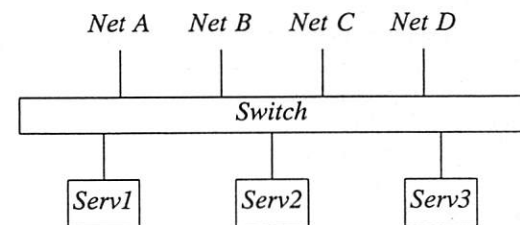
### A Switch-based Server Network

We had already looked at Ethernet switches for other purposes, but now began to study them as an alternative to FDDI and simple Ethernet for our server network. They offered the advantages of Ethernet for the server connections – low cost per server with thoroughly tested hardware and software – with much higher aggregate bandwidth across the network.

Out of several possible choices, we found the Alantec PowerHub's unique routing capabilities [16] to be intriguing. Our original idea for a switch-based server network treated routers just like servers, as in Figure 3a. The Alantec allowed us to skip routers when going to the most critical networks, leading to the network structure in Figure 3b. (Routers are still used for non-IP traffic and for networks which are not directly connected to the switch; they were omitted from the diagram for clarity.)



3a. routers to client networks



3b. switch does routing to client networks

Figure 3: Switch-based network topologies

The initial PowerHub 3500 does not support enough ports to dedicate one to each server, but careful balancing of servers amongst the available ports makes this tolerable. Eliminating the need for this manual balancing will likely make the PowerHub 7000 [17] an attractive future upgrade.

### Performance Benefits of Switched Ethernets

Every node that is added to the collision domain of a CSMA/CD network (such as Ethernet) cuts the available bandwidth for the other hosts on the network two-fold. The new node cuts out a share of the bandwidth for its own communication, then takes another cut because increasing the number of nodes on a broadcast network decreases the maximum throughput possible – two nodes communicating on an Ethernet can transmit at nearly the full 10 Mb/sec available, but an Ethernet that has many nodes will see the effective throughput of the medium drop to about 4 Mb/sec.

A switched Ethernet allows for aggregate throughput to increase everytime a node is added. There is a limit, of course, and it is dependent on the backplane internal to the switching hub, usually at least several hundred Mb/sec. (The Alantec PowerHub 3500 selected by SLAC has a 400 Mb/sec backplane.) Unlike a 10BaseT hub, a switched Ethernet hub supports a separate collision domain on each segment attached to it. If only one node is connected to a port, it has the capability of transmitting or receiving data at a full 10 Mb/sec. Thus, adding another node to a second port adds another 10 Mb/sec to the aggregate bandwidth, and so on for each additional connection.

### Other Advantages of Switched Ethernets

At first glance, the price of a switched network seems prohibitive. Installation of non-switched LANs start at about \$200 per node for a 10BaseT hub, then can quickly climb to \$1,000 per node for switched Ethernet, and \$2,500 per node for FDDI [18]. However, this perspective ignores performance considerations. Factoring in the bandwidth of the network, switched Ethernet becomes very attractive at only \$100 per node  $\times$  Mb/sec ( $N \cdot \text{Mb/sec}$ ), followed by FDDI at \$250 per  $N \cdot \text{Mb/sec}$ , and finally 10BaseT at \$600 per  $N \cdot \text{Mb/sec}$ .

The isolation between different Ethernet segments afforded by switches can also reduce the likelihood of interoperability problems such as those described in [7]. Store-and-forward switches such as the Alantec offer more isolation than cut-through designs such as Kalpana's EtherSwitch, at the cost of higher latency [19].

### Disadvantages of Switched Ethernets

Switches are not without cost, however. Truly dedicated ports preclude Ethermeters or other

monitoring devices, and even if they didn't, the cost of an Ethermeter for each port would be prohibitive. It may be possible to send all or selected traffic to a designated monitoring port, but this defeats much of the point of switches, and if the network is busy the monitoring port will surely be flooded. Sending only selected data to the monitoring port may keep the load down, but precludes any on-going monitoring and automated problem detection. What's really needed is for the switch itself to provide full RMON data for each port.

A switch also represents a single point of failure, a grave concern for a server network which is critical to most of an organization. A coaxial cable may not have the bandwidth of an Ethernet switch, but it also doesn't have power supplies and software which can fail. Some switch vendors are addressing these concerns by offering redundancy and hot-swappable power supplies and other modules.

Adding ports may be another hurdle. Kalpana's solution is to cascade EtherSwitches, but this creates a potential bottleneck in the Ethernet between switches. Alantec's 3000 and 5000 models are somewhat limited in the number of ports they can support, but multiple switches can be chained together using FDDI. While this offers higher bandwidth than the Kalpana solution, the marginal cost of the next port after all ports on the first Alantec switch have been used is exceedingly high. Fortunately, newer switches are appearing with dramatically greater capabilities for port expansion.

Finally, the process of selecting a switch is difficult, since each switch seems to have a remarkably different feature set.

### FDDI Still Has a Place

Switches can be useful tools, but there remain network applications where they cannot substitute for high-speed networks such as FDDI or 100 Mb/sec Ethernet. The computing environment for the next major experiment at SLAC, the Asymmetric B Factory, and our planned T3 (45 Mb/sec) connection to the Internet are two such examples.

The B Factory will involve several hundred terabytes (tera =  $10^{12}$ ) of data by the end of the experiment. This data will be stored in a complex of StorageTek silos, and off-line analysis will be done by a farm of workstation-class machines, as illustrated in Figure 4. To optimize use of the tape drives, tape data will be staged to disk. Many of these data paths individually require FDDI speeds, and the aggregate speed of the network well exceeds FDDI, so a DEC Gigaswitch will be employed as the backbone. Each port on the Gigaswitch will in effect be its own FDDI ring, reducing concerns about FDDI interoperability. The compute servers in the farm do not individually require high bandwidth,

so they will be connected with Ethernet to an Alantec switch, which in turn will connect to the Gigaswitch via FDDI.

The upgrade of SLAC's primary Internet connection from T1 to T3 provides a simpler, and slightly more commonplace, example of the need for networks with FDDI speeds. An Ethernet can readily handle traffic at a T1 line's 1.544 Mb/sec, but a T3, at 45 Mb/sec, is far beyond the ability of an Ethernet. The firewall router, which as seen above already contributes a sizeable amount of traffic to the server network, will be replaced with a larger router connected directly to an FDDI ring. From there, it will be able to send packets to various internal routers and to the B Factory compute farm at full T3 speed. Except for the Gigaswitch, all devices on this FDDI will be Cisco routers, so interoperability concerns are again minimal.

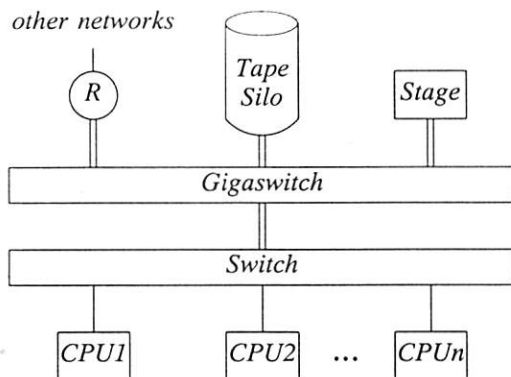


Figure 4: Compute farm for Asymmetric B Factory

### Conclusions

FDDI can handle high traffic volumes to and from a single server, but is expensive and still not mature. In a network with a number of smaller servers handling clients on a variety of networks, aggregate bandwidth of the server network may be more important than the capacity of the connection to individual servers, in which case an Ethernet switch offers higher bandwidth at a lower cost and with fewer potential interoperability problems. Switches are not a universal solution, however – FDDI still has a place where high bandwidth to a single server is required. Good monitoring of the network to characterize traffic patterns is invaluable for choosing the best solution.

### Acknowledgments

Thanks to Bob Cook, Charlie Granieri, Diana Gregory, Krissie Griffiths, Connie Logg, Jo Ann Malina, Alexander Shaw, Lois White, and others whose contribution is not diminished by our failure to credit them. Our gratitude goes to all of them.

### Author Information

Karl Swartz is a member of the Server Systems Group within SLAC Computing Services at the Stanford Linear Accelerator Center, where he is the resident UNIX guru. Prior to joining SLAC, he worked at the Los Alamos National Laboratory on computer security and nuclear materials accounting, and in Pittsburgh at Formtek, a start-up that is now a subsidiary of Lockheed, on vector and raster CAD systems. He attended the University of Oregon where he studied computer science and economics. Karl instructs at high-speed driving schools and enjoys good food and good beer (though not while driving) and hikes on the beach with his Golden Retriever, Alexander. Reach him via electronic mail at [kls@chicago.com](mailto:kls@chicago.com) or [kls@slac.stanford.edu](mailto:kls@slac.stanford.edu), or see his Web page at <http://www.slac.stanford.edu/~kls/kls.html>.

Les Cottrell left the University of Manchester, England in 1967 with a Ph.D. in Nuclear Physics. He joined SLAC as a research physicist focusing on real-time data acquisition and analysis. In 1972/73 he spent a year's leave of absence as a visiting scientist at CERN in Geneva, Switzerland, and in 1979/80 at the IBM UK Laboratories at Hursley, England. He is currently Assistant Director of SLAC Computing Services and focuses on networking and distributed computing technologies. Reach him via U.S. Mail at Mail Stop 97, SLAC, P.O. Box 4349, Stanford, California, 94309. Reach him via e-mail at [cottrell@slac.stanford.edu](mailto:cottrell@slac.stanford.edu), or see his Web page at <http://www.slac.stanford.edu/~cottrell/cottrell.html>.

Marty Dart is a Network Engineer at SLAC where he previously worked as a Technical Coordinator and Network Technician. He received his BSEE from San Francisco State University in 1988. Marty can be reached via e-mail at [dart@slac.stanford.edu](mailto:dart@slac.stanford.edu), or via U.S. Mail at Mail Stop 97, SLAC P.O. Box 4349, Stanford, California, 94309.

### References

1. Karl L. Swartz, "Optimal Routing of IP Packets to Multi-Homed Servers," *Proceedings of the 6th USENIX Large Installation System Administration Conference (LISA VI)*, pp. 9-16, Long Beach, October 1992. Also published as SLAC-PUB-5895.
2. Hal Stern, *Managing NFS and NIS*, O'Reilly & Associates, Inc., Sebastopol, California, 1991.
3. Art Wittmann, "An FDDI Primer: Riding the Photons," *Network Computing*, pp. 134-144, January 1993.
4. *IBM FDDI Workstation Adapters*, IBM Product Announcement 192-132, IBM Corp., May 19, 1992.

5. Todd Tannenbaum and Michael Lee, "FDDI Adapters Take The SBus To High Performance," *Network Computing*, pp. 108-110, April 1, 1994.
6. Bob Metcalfe, "From The Ether," *InfoWorld*, November 15, 1993.
7. Wesley Irish, "High utilization Ethernet performance problems traced to controller," *comp.dcom.lans.ethernet (Usenet newsgroup)*, October 15, 1993.
8. Connie Logg and Les Cottrell, "Adventures in Network Performance Analysis," talk at the *1994 IEEE Network Operations and Management Symposium*, Kissimmee, Florida, February 1994.
9. Connie Logg and Les Cottrell, "Network Performance Monitoring and Analysis at SLAC," talk at the *1994 Dept. of Energy Telecommunications Conference*, Baltimore, August 1994.
10. Lyle Seaman, "AFS Performance Evaluation: Observations, Analysis, and Plans," *Proceedings of DECORUM '94*, Orlando, March 1994.
11. *AFS 3.3 Release Notes*, p. 51, Transarc Corp., Pittsburgh, January 1994.
12. David R. Boggs, Jeffrey C. Mogul, and Christopher A. Kent, "Measured Capacity of an Ethernet: Myths and Reality," *Computer Communications Review*, vol. 18(4), pp. 222-234, ACM, August 1988.
13. Michael Lee and Art Wittmann, "Four Steps To Better FDDI," *Network Computing*, pp. 140-141, April 1, 1994.
14. Peter Hoogenboom and Jay Lepreau, "Computer System Performance Problem Detection Using Time Series Models," *Proceedings of the USENIX Summer 1993 Technical Conference*, pp. 15-32, Cincinnati, June 1993.
15. Gary L. Schaps and Peter Bishop, "A Practical Approach to NFS Response Time Monitoring," *Proceedings of the 7th USENIX Large Installation System Administration Conference (LISA VII)*, pp. 165-169, Monterey, California, November 1993.
16. *PowerHub Reference Manual*, Alantec, July 1993.
17. Skip MacAskill, "Alantec powers up new high-end switching hub," *Network World*, p. 1, 56, July 18, 1994.
18. Peter Sevcik and Mary Johnston Turner, "Enterprise Networks: Architecture and Planning," tutorial at *Networld+Interop 94*, Las Vegas, May 1994.
19. J. Scott Haugdahl, "Switch trio boosts bandwidth," *Network World*, pp. 47-54, July 25, 1994.

# Pong: A Flexible Network Services Monitoring System

*Helen E. Harrison, Mike C. Mitchell, and Michael E. Shaddock – SAS Institute, Inc.*

## ABSTRACT

In distributed computing environments it is important to determine not only whether individual critical machines are up or down, but also whether the individual services they offer are available. Our site was using a network monitoring package which used ICMP ECHO packets to determine when individual network components were unavailable. We found that there were many occasions where a server would reply to the ICMP ECHO, but would still not be providing the services it should be providing. We needed a tool which would let us monitor high level services such as AFS or NFS file service. This paper will describe a tool called pong which was developed to meet these needs. Pong is a highly configurable monitoring tool which "pings" individual services at predefined intervals and executes appropriate actions when the state of that service changes. We use pong to monitor three or more services on each of 110 servers.

## Introduction

SAS Institute Inc. is a privately owned software development company whose primary software development environment is a network of 1500 Hewlett Packard 700 series workstations and servers [1]. These machines share data using a combination of AFS and NFS file systems over an aggressively subnetted TCP/IP network. During the second half of 1992 we experienced increasing occurrences of failures in services on key servers which were not being detected by our commercial network monitoring package. The monitoring package used only ICMP ECHO packets to determine if a machine was operational. We found that the use of ICMP ECHO packets never gave a false negative, but did give false positives. The HP servers would occasionally fail in a way that they would still respond to pings, but otherwise would appear to be unavailable. In addition we found situations where the AFS or NFS daemons would stop responding to requests but otherwise the machine would be healthy. We needed a monitoring system which would allow us to determine the state of a system and the services it was providing. Another requirement we have for a monitoring system is that it be configurable and easily extensible. For example, when a failure is detected we may want to send mail, page someone, send a zephyr-gram, log the failure, or whatever else we might decide we need. We want to control how often a check is performed, what type of check, and how many replies are "dropped" before a service is considered unavailable. Pong addresses these requirements. There have been number of other projects which also addressed these problems, including the emerging SNMP based technologies, and many of these show great promise. So far these alternatives have not provided the functionality, efficiency, and cost

effectiveness that would make them appropriate replacements for pong for service level monitoring.

## The Pong Program

Pong has three basic characteristics: service monitoring routines, monitoring intervals and response actions. Pong monitors these services by constructing a UDP packet and sending it to the appropriate port on the remote machine and then watching for a response. The service monitoring routines include ICMP ECHO requests which measures basic network connectivity, the `inetd` "echo" service which measures `inetd`'s responsiveness (a good measure of the overall health of a machine), NFS `nullRPC` messages which measures NFS responsiveness, and AFS `RX DEBUG` requests which measures AFS's responsiveness. These functions simply return success or failure. Each of these built-in monitoring functions were implemented using existing functionality and required no changes on the systems being monitored. Response actions are executed when a machine changes state. Pong does not have any built-in notification routines. Its sole job is to monitor the state of the specified machines leaving notification of state changes to other programs.

If a monitored service becomes unresponsive, pong starts checking with increased frequency. After the specified number of "down" checks are performed and the service is still unresponsive, pong executes the specified "down" response action. It will then switch back to checking less often, until the service responds. After the first positive response it starts checking more often until the specified number of "up" checks are performed. At that time it executes the specified "up" response action, and switches once more to checking less often (see Figure 1).

## Configuration

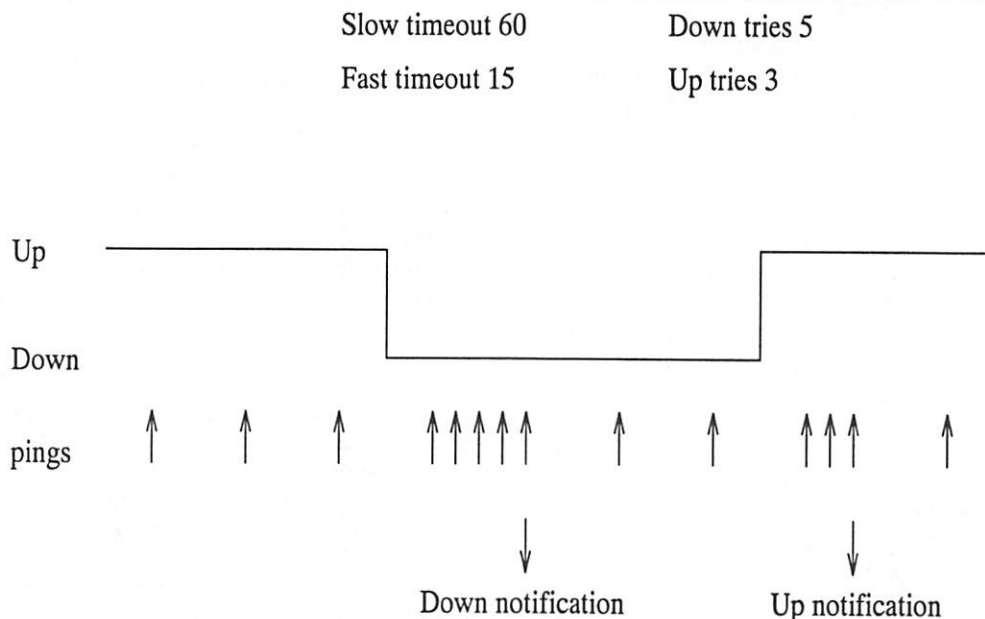
The pong configuration file lists the type of service to monitor (icmp, inetd, nfs, or afs), the frequency with which to check the service, the command to execute when the service is discovered to be unavailable, the command to execute when the service is available again, and finally the list of machines on which to monitor these services (see Example 1).

The first line of the configuration file specifies the type of service, the time between checks when it knows the service status, the time between checks when it isn't sure of the status, the number of consecutive lost responses before a service is marked down, and the number of consecutive received responses before a service is marked up. The next two lines list the command to be executed when the the service is marked down and marked up, respectively. The following lines list all of the machines this type of check will be performed on. It will

accept hostnames, IP addresses, and hostclass specifications [2].

The up and down command lines will be passed to `/bin/sh`, so they can full advantage of UNIX regular expression functionality. Pong replaces “%h” on the command line with the host name (or IP address) of the failing service. The “%t” and “%T” character sequences are replaced with the time the service went down and the current time, respectively. The time is converted to the UNIX `ctime` format, without the newline on the end (see `ctime(3)` manual page).

The usefulness of pong depends greatly on the scripts used for notification. For simplicity this example shows the down command as an echo command piped to a mailer. For our production system we actually have pong execute a perl script. Our perl script writes an informative timestamped line to a log file, sends mail to the system administrator, sends a color-coded Zephyr notification[3] (red for



### Figure 1: Polling Intervals

```
#
# nfs pings -- send a null rpc to NFS process.
#
#          slow      fast      down      up
#type      timeout  timeout  tries      tries
nfs        60       5         4         3
echo down "%h: nfs down at %t" | /bin/mail sysadmins
echo up    "%h: nfs up at %T" | /bin/mail sysadmins
dbserve1
152.5.64.10
=code servers
```

**Example 1:** A typical entry in the configuration file

"down" messages and green for "up" messages) to an interested party subscription list including our Data Center operators and the systems administrator it has determined to be "on call". If that system administrator is not presently logged in, the notification routine will send him mail. We send our messages to the "servdown" Zephyr class, using the machine name as an instance. (The Zephyr system manages subscription lists and sends out messages to anyone who subscribes to a specific "class" and "instance", somewhat similar to newsgroups or mailing lists.) The use of Zephyr allows people who are interested in particular servers to receive notification without requiring intervention by support personnel.

### Extending Service Monitors

As described previously, pong currently monitors four basic services (NFS, AFS, inetd and ping) and we plan to extend it to monitor other services such as named and zephyrd. Pong is structured so that additional service definitions can be added easily by adding entries to a table of functions. Pong constructs a UDP packet based on the function definition and sends it to the specified port. All that is needed is to identify a way to probe a network service so that it sends a packet in response. For example, AFS does not have a predefined ping-type function like the inetd echo server. In order to "ping" AFS, we construct an invalid AFS RX DEBUG packet to which the AFS server process responds with a packet that says "this request is invalid". If there is not a way to probe a service via a UDP packet, it can be added to pong by defining a service under inetd that queries the desired service, and have Pong poll the new inetd service.

When Pong starts it initializes an event queue which contains one entry for each machine and ping type combination. The entries also contain a subroutine to call and the time to make the call. After the

event queue is initialized, Pong loops calling select and handling events. The top of the loop checks to see if an event timer has expired, and if so calls the appropriate function. After it handles all the expired timers, it calculates the amount of time until the next timer expires. It passes that time off to the select routine as the maximum time to wait for input on a file descriptor. If the select routine indicates there are packets ready to be read, the appropriate input routines are called. After all the packets are read the loop returns to the top.

There are only three subroutines that need to be written to add a new "ping" type. Pong calls an initialization routine at startup, a packet-send routine when a timer expires, and a packet-read routine when the select call says there is a packet to be read.

The initialization routine is passed a pointer to a "ping\_type" structure, in which it fills in a file descriptor. Anything else it does is dependent on what type of "ping" is being created. Typically the initialization routine opens a socket, looks up what UDP port number to use, and fills in whatever data structure will be used as a packet header. Information that might be needed to send a packet or receive a packet can be stored in global variables.

The packet-send routine is called every time a packet of the appropriate ping-type is to be sent to a host. It is passed a pointer to some host-specific data which includes the IP address of the host, a pointer to the "ping\_type" structure (which contains the file descriptor), and a unique sequence number for that packet. Typically all the packet-send routine does is put the sequence number into the packet built by the initialization routine, fill in the socket address with the UDP port number and host address, and send the packet.

The packet-receive routine is called after the select call indicates there is a packet to be read. It is passed a pointer to the "ping\_type" structure

```
% pongstat nastase
Host      Ping    State      Last Up      Last Down
-----
nastase   afs     down
nastase   inetd   up        Sun May 22 15:25:46 1994
nastase   nfs     up        Sun May 22 15:25:11 1994
nastase   Sun May 22 09:02:47 1994

% pongstat -d
Host      Ping    State      Last Up      Last Down
-----
seles     nfs     down
nastase   afs     down
seles     Sun May 22 10:54:44 1994
nastase   Sun May 22 09:02:47 1994

% pongstat -help
usage: pongstat [-p server] [-t type] [-d] [ machine ... ]
```

Example 2: Sample output of the pongstat command

(which has the file descriptor). Typically a packet is read from that file descriptor and the return address is used to look up the host-specific data. Once the host-specific data has been located, the packet-receive routine can check the received packet to be sure it has the right sequence number, the right format, and so on. If the reply packet is acceptable, the packet-receive routine sets a flag in the host-specific data structure indicating that it received a reply from the packet sent.

A ping-type that uses a virtual circuit instead of datagrams (TCP instead of UDP) would be possible, even though pong was designed for datagrams. For a TCP based "ping", both the initialization routine and the packet-receive routine would be empty stubs. The packet-send routine would open the socket, make the connection, exchange whatever information is necessary, and close the connection. If the packet exchange indicated the service was functioning, the packet-send routine would set the special "I received a reply" flag in the host-specific data structure. The problem with this approach is that it can take a long time for the TCP connection to be established and a long time for the packet exchange. The person writing the code must put in code to abort the connection if it takes more than a few seconds for the the reply to come back, otherwise all of the other pings will be delayed.

### Related Utilities

#### Pongstat

Once we had pong up and running, we found that we wanted to be able to determine current state of all the servers and services it was checking. Pong writes messages to a log file, but it was not practical to parse through all the messages to find the current status. Pong knows the state, so we modified it to dump out the current state when it received a TCP connection to the pongstat port. We could then use a command like telnet to get a current status report, but we decided it would be simpler to write another program to get the information; we call it pongstat.

Pongstat uses TCP to connect to the pong program. Pong sends it an alphabetized list of machines, services, an up/down flag, and the time of the last down-to-up and up-to-down transitions. Pongstat reformats the output and prints out only the services and/or machines specified. By using the "-d" flag, only the services that are currently down are shown. The sample output in Example 2 shows that a server named nastase is up except for its AFS processes. Everything on that system went down around 9:00 AM and came back up 15:25. The only things currently down are NFS on seles, and AFS on nastase.

#### King\_Pong

The pongstat command served our system administrators well, but the our operators wanted a simple way to tell the state of the network at a glance. From that need came king\_pong, a GUI program that graphically displays the current state of all the servers. It is written using the Tcl, Tk, and Extended Tcl packages. It connects to pong once a minute via the pongstat port and draws red, green and yellow boxes showing the server status. A green box means that a server is up; a red box means that a service on that servers is down; a yellow box means that a service has gone down and back up again. A yellow state must be acknowledged before returning to green. This ensures that failure conditions which correct themselves will not go unnoticed.

A mouse click on a box pulls up a window that shows the current state, as if pongstat was executed for that particular machine. In addition to getting server status with a mouse click, it also has the ability to display additional information about each server, such as which groups use it, network services which it provides, its physical location and special contacts for the server. This information allows us to give general documentation to operations on how to respond to alarms while keeping the more dynamic specific information on each server in a more easily updated online form. King\_pong also allows the operators to ignore a server, so that if a server needs to be out of production for a couple of days, they will not be seeing alarms while it is out of service.

### Related Work

Pong has similarities to the buzzerd system [4] in implementation and solves similar problems. Through notification programs, pong can be configured for notification schemes of similarly complex to buzzerd. In our environment, we have a Data Center which is staffed 24 hours/day so that notification programs could be fairly simple. Instead our efforts were concentrated on flexibility of its monitoring capabilities. One significant difference is that buzzerd requires that a special daemon be run on a each server that it monitors. Pong does not require that anything be changed on a monitored machine which can be a distinct advantage if the target is a PC, router, terminal server, network printing device, or other network device which does not happen to run UNIX. With over 100 servers to monitor, we do not want the overhead of managing an extra monitoring daemon if we can avoid it.

For situations which would require running a special monitoring daemons on each machine, pong could still be used as the notification agent. The monitoring daemon could check many different system resources like the sysmod component of buzzerd, or even be more sophisticated like the SPA

Expert System[5]. As long as the monitoring daemon can respond to pong's queries, pong can be used as part of the notification process. Using one notification agent simplifies administration of the many different monitoring tools.

If a network router or bridge fails, pong may try to send 100's of "down" messages, even though only a single component is really unavailable. The "Big Brother" system[6] addresses this type of problem by introducing dependency lists in its configuration file. With pong this problem can be addressed by using "smart" notification scripts. There are some interesting trade-offs here, which for pong were decided in favor of simplicity.

Many of the commercial network monitoring systems are using SNMP as their querying mechanism. Pong does not use SNMP for several reasons: SNMP is not universally available, SNMP is a different protocol with a learning curve of unknown steepness, and SNMP would have required another daemon on each monitored machine. Instead, it was decided to use well known and well understood protocols. Given the extensible nature of pong, we believe it will be relatively simple to add SNMP querying functions.

### Conclusions

Commercial network monitoring systems have been emerging over the last few years which show promise of becoming highly sophisticated and useful. Unfortunately they have not reached their full potential and have been expensive for the core functionality that they have offered. Specifically, we have found no commercial tools which will monitor AFS services. Pong was written as a survival tool to fill this vacuum and help us monitor our rapidly growing network. Its simplicity and flexibility have made it a powerful tool.

### Availability

For information on the availability of Pong, please send mail to [heh@unx.sas.com](mailto:heh@unx.sas.com).

### Author Information

Helen E. Harrison is the UNIX Support Manager at SAS Institute Inc., where her group provides hardware and software support for a network of over 1400 UNIX workstations and servers. She has been involved in UNIX systems administration for over 10 years and holds a B.S. in Computer Science from Duke University. Reach Helen at SAS Institute Inc, SAS Campus Drive, Cary, NC 27513; or by e-mail at [heh@unx.sas.com](mailto:heh@unx.sas.com).

Mike Mitchell is a Systems Programmer in the UNIX Support Group at SAS Institute Inc. He has been involved in Distributed Computing for over 5 years and UNIX systems for 15 years. He holds a B.S. in Computer Science and a B.S. in Electrical

Engineering, both from North Carolina State University. Reach Mike at SAS Institute Inc, SAS Campus Drive, Cary, NC 27513; or by e-mail at [mcm@unx.sas.com](mailto:mcm@unx.sas.com).

Michael Shaddock is a Systems Programmer in the UNIX Support Group at SAS Institute Inc. He has been involved in UNIX systems administration for over 8 years and holds a M.S. in Computer Science from the University of North Carolina at Chapel Hill. Reach Michael at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513; or by e-mail at [shaddock@unx.sas.com](mailto:shaddock@unx.sas.com).

### References

1. Helen E. Harrison, "So Many Workstations, So Little Time," *LISA VI Proceedings*, pp. 79-86, Long Beach, CA, October, 1992..
2. Helen E. Harrison, Stephen P. Schaefer, and Terry S. Yoo, "Rtools: Tools for Software Management in a Distributed Computing Environment," *Proceedings of the Summer USENIX Conference*, pp. 85-93, San Francisco, CA, June, 1988..
3. C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld, "The Zephyr Notification Service," *Proceedings of the USENIX Winter Conference*, pp. 213-219, Dallas, TX, February, 1988..
4. Darren R. Hardy and Herb M. Morreale, "buzzer: Automated Systems Monitoring with Notification in a Network Environment," *LISA VI Proceedings*, pp. 203-210, Long Beach, CA, October, 1992..
5. Peter Hoogenboom and Jay Lepreau, "Computer System Performance Problem Detection Using Time Series Models," *Proceedings of the Summer USENIX Conference*, pp. 15-32, Cincinnati, Ohio, June, 1993..
6. Don Peacock and Mark Giuffrida, "Big Brother: A Network Services Expert," *Proceedings of the Summer USENIX Conference*, pp. 393-398, San Francisco, CA, June, 1988..

## Appendix A: Sample Notification Script

```
#!/local/bin/perl
$host = $ARGV[1];
$type = $ARGV[2];
$dtm = $ARGV[3];
$ctm = $ARGV[4];
$log = "/local/etc/servdown.log";

if ( $ARGV[0] eq "up" )
{
    $color = "green";
    $msg1 = $host . ": " . $type . " up at " . $ctm;
    $msg2 = "down at " . $dtm;
}
else
{
    $color = "red";
    $msg1 = $host . ": " . $type . " down at " . $dtm;
    $msg2 = "x";
}

open(LOG, ">>$log");
print(LOG "$msg1\n");
close(LOG);

if ( $msg2 ne "x" )
{
    open(MSG, "|/local/bin/zwrite -n -c servdown -i $host");
    print(MSG "@beep()@color($color)@large(@b($msg1))\n");
    print(MSG "@beep()@b($msg2)\n");
    close(MSG);
}
else
{
    open(MSG, "|/local/bin/zwrite -n -c servdown -i $host");
    print(MSG "@beep()@color($color)@large(@b($msg1))\n");
    close(MSG);
}

$onc = "no_one";
open(ONC, "/local/etc/pong.oncall");
while(<ONC>)
{
    chop;
    if (length($_) > 0)
    {
        $onc = $_;
        close(ONC);
    }
}
close(ONC);

$rc = 1;
if ($onc ne "no_one")
{
    $rc = system("/local/bin/zlocate $onc > /dev/null 2>&1");
    if ($rc == 0)
    {
        open(MSG, "|/local/bin/zwrite -n $onc");
        print(MSG "@beep()@color($color)@large(@b($msg1))\n");
        if ( $msg2 ne "x" )
    }
}
```

```
{
    print(MSG "@beep()@b($msg2)\n");
}
close(MSG);
$rc = $?;
}
if ($rc != 0)
{
    open(MSG, "|/usr/bin/mailx -s \"server problems\" $onc");
    print(MSG "$msg1\n");
    if ( $msg2 ne "x" )
    {
        print(MSG "$msg2\n");
    }
    close(MSG);
}
}
```



# Automating Printing Configuration

Jon Finke – Rensselaer Polytechnic Institute

## ABSTRACT

Maintaining the printing configuration files for a large site (400 machines, 60 public printers, 40 private printers, 30 print spoolers) can be a major job. At RPI, we developed a system that will automatically generate the printer configuration file for any machine, depending on what printers are driven by that machine. It also allows us to only have a printer appear only on a subset of machines, rather than on all machines.

This paper describes the design and deployment of the system. We use a relational database to manage the printer information, printer type attributes, host grouping and to track hosts using the system. All sources and related information are available for anonymous FTP.

## Motivation

One of the first parts of the Simon project<sup>1</sup>, was a program called *printmaster*, which was used to generate the */etc/printcap* file for any of our workstations or print servers.<sup>2</sup> While it did have attribute matching for printer types, it did not support host groups, and in fact did not even have a UI to make changes. All configuration info was entered using SQL<sup>3</sup> scripts via the Oracle *SQL\*Plus* program. An even bigger problem, was that the configuration generation program ran on a central machine and "pushed" the configuration out to the client host. What finally killed this version, was that we had lost source to one of the critical libraries, so we were unable to make changes in the program.

Despite some of the operational problems with the original program, the basic design was solid, and a few years of operational experience made it very easy to update the design and re-implement it (with a few improvements) on our production systems. We considered going to a double entry approach suggested by David Wood[1], but decided that the ability to restrict printers to just a subset of hosts, as well having it fit with our general system administration model ruled the day and we did the rewrite.

This resulted in three new programs, *etcprintcap* which generates the actual *printcap* file and installs it if needed,

<sup>1</sup>Simon is a project we are developing to automate many aspects of computer system administration. It relies heavily on the use of a relational database to manage user, host, and other types of data, and allow later modules to build effectively on earlier work. Simon has been described in a number of papers and presentations, see the *Availability* section for details.

<sup>2</sup>We use a somewhat customized version of the Berkeley *lpr* print spooling programs for both SunOS and AIX.

<sup>3</sup>SQL is the language used by Oracle and other relational databases for specifying queries and other functions.

*printmaster* which provides a user interface to edit printers and printer attributes, and finally,

*printcap\_status* which reports on what version *printcap* file each host has, and when that host last checked in.

We now have the *etcprintcap* program running on all of our spooler machines via cron every night. We also have another cron job on the database machine that refreshes the common *printcap* file that the rest of the generic machines use. This has made it quite easy to add new printers to the system. Just run *printmaster*, answer a few questions (a few more if it is a new printer type), and by the next morning, the correct definitions are installed on all of the machines.

## Printers and Printer attributes

The *printmaster* program is used to maintain the two primary tables, **PRINTERS** and **PRINTER\_ATTR**. In addition, a third table **PRINTCAP\_LOG** is used to record when a host gets a new *printcap* file, and the last time it has checked the database. These tables are described below. We also make use of some of the Hostmaster tables and some host grouping tables that have been described in other papers.[2]

Each printer has certain characteristics that are unique to that specific printer, such as the name, the spooler machine, the type of printer, the type of connection to the spooler, the host group and so on. This information is stored in the **PRINTERS** table.

## PRINTERS Table Definition

**id number** A Simon.Peoplecount.Nextval<sup>5</sup> that uniquely identifies this printer. It is used when other tables need to join to a printer name.

<sup>4</sup>Many local files on our machines are maintained out of a common file system (AFS) via package.

**longname char(36)** The long form of the printer name. This is the name that is generally used in displays and usually defines for people, what and where the printer is.

**shortname char(8)** A short form (8 characters or less) of the printer name. This is generally used when a person has to enter the printer name by hand. This name is also used for the spool and log directory trees (as defined in the attributes).

**spooler number** The Simon.Dns\_Domains.Id<sup>5</sup> of the host that drives or spools for the printer. This is used to find the hostname for "rm=" entries and when generating printcap files on that host.

**printer\_type char(8)** The type of printer being driven. This is matched with the Print\_Attr.Printer\_Type field when attributes are generated. In practice, we use generic printer types to avoid duplicating too many attributes.

**connection\_type char(8)** The type of connection used to connect the printer to the spooler. This is only relevant on the spooler machine. For attribute matching, all remote printers are assumed to have a connection type of "LPD". This is matched to Print\_Attr.Connection\_Type.

**per\_page\_charge number** The per page charge for the printer in centi-cents. This can be included in the printcap file, and should also be used by the printer accounting software.

**per\_job\_charge number** The per job charge for the printer in centi-cents. This can be included in the printcap file and should also be used by the printer accounting software.

**budget number** The Index to the budget that is credited with all charges from this printer. This is used by the printer accounting software. Values here are defined by the accounting system.

**spool\_from number** Is the Simon.Dns\_Domains.Id of the single host, or Simon.Host\_Groups.Group\_Id of the host group that contains all the hosts that should have this printer in their printcap file. This allows a printer to only appear on a subset of all hosts.

**when\_inserted number** The Simon.Transcount.Nextval of when this record was inserted into the table. This is used to help set the database version.

<sup>5</sup>A *object*.Nextval refers to the next value of a sequence named *object*. Each time *object*.Nextval is referenced, it returns the next value in the sequence (generally, increment by 1.) These are maintained automatically by Oracle.

<sup>6</sup>The format *Object.Member* is generally used to indicate that we are referring to column *Member* of table *Object*. In this case, we are referring to the Id column of the Dns\_Domains table.

**when\_updated number** The Simon.Transcount.Nextval of when this record was last changed. This is used to help determine the database version number.

**when\_marked\_for\_delete number** The Simon.Transcount.Nextval if when this record is considered obsolete and should be ignored. This is how we can delete a printer.

**comments char(240)** A space to include a single line of comments on a printer. These comments, if any, will be included as comments in the printcap file.

**system\_id number** A Simon.Systems. Id value that can be used to assign location, owner, sys admin and other systems and related attributes to this printer.

**when\_attributes\_updated number** A Simon.Transcount.Nextval of when an attribute that matches the id, printer type and connection\_type of this printer is updated. This is used to trigger a printcap regeneration when just an attribute of a printer changes.

Most of the text that goes into the actual printcap file, is extracted from the PRINTER\_ATTR table, which allows matching by specific printer, and by combinations of printer type and connection type.

#### PRINTER\_ATTR Table definition

**printer number** The Simon.Printers.Id of the specific printer that this printcap entry is for. If null, connection type and printer type are used for matching.

**printer\_type char(8)** An optional printer type used to match with Simon.Printers.Printer\_Type.

**connection\_type char(8)** An optional connection type use to match with Simon.Printers.Connection\_type. During printcap generation, all remote printers are assumed to have the type "LPD".

**rank number** A number used to rank order lines in the printcap file, and to allow more specific attribute matches to "override" more general attribute matches. Each different printcap keyword has a specific rank.

**entry char(120)** The actual entry to be included in the printcap file. A number of macro strings are defined by the etcprintcap program that will be expanded at printcap generation time.

**when\_inserted number** The Simon.Transcount.Nextval of when this entry was inserted into the table.

**when\_updated number** The Simon.Transcount.Nextval of when this entry was changed in some way.

**when\_marked\_for\_delete number** The Simon. Transcount.Nextval of when this entry is considered obsolete and no longer included in the printcap file.

Consider the following selection from the **PRINTERS** table in Figure 1 (with spooler\_id replaced by the actual spooler name).

Short Name	Printer Type	Link Type	Spooler Name
AE325LW	LW	SER	beale.math.rpi.edu
AE217LW	LW	SER	buster.math.rpi.edu
VC215LW	LW	CAP	netserv2.its.rpi.edu

Figure 1: Printers table excerpt

And the following selection from the **PRINTER\_ATTR** table in Figure 2. (note: A (null) indicates that no value is set for that particular data element. In this case, it means "don't care".):

We order the attribute lines by the rank value, and by the "closeness" of the match. This will let an attribute for that specific printer override an attribute that only matched for the printer type. This allows us to override one particular line for a given printer, and still have the printer get the standard

entries for that particular type. The rank ordering also allows us to maintain a consistent order in the printcap entries. There is also a special link type, "LPD". This is used when matching for printers that are NOT driven by the current host. Assuming we were on beale.math, we would get a printcap file like the one in Figure 3 and for the printcap on buster.math, one like Figure 4.

We can see a couple of special cases here. The VC215LW printer is located on a host *outside* of the systems we administer, and that printer name does not follow the pattern we use. We can override the default remote printer name ("rp") with a specific entry for just this printer. We did a similar thing for the "lp" name for AE217LW.

### Printcap generation

With at least a rough idea on how the attribute matching works, lets take a step back and look at the whole process of generating and installing a printcap file. All of our hosts periodically (usually once a day), run a program to update the system configuration and software. On a subset of these, the *etcprintcap* program is run.

Short Name	Printer Type	Link Type	Rank	Entry
(null)	LW	SER	110	lp=/dev/{printer_name}
AE217LW	(null)	SER	110	lp=/dev/ttya
(null)	(null)	(null)	120	sd=/usr/spool/{printer_name}
(null)	(null)	LPD	150	rm={spooler_name}
(null)	LW	LPD	160	rp={printer_name}
VC215LW	LW	LPD	160	rp=sparclw

Figure 2: Printer\_Attr table excerpt

```
#
# System printcap file for beale.math.rpi.edu.
# Generated by SIMON on 17-MAY-94 from
# printer DB version 6358435
#
AE325LW|Amos_Eaton_325_Private_LaserWriter:\
:lp=/dev/AE325LW:\
:sd=/usr/spool/AE325LW:...

AE217LW|Amos_Eaton_217_Private_LaserWriter:\
:sd=/usr/spool/AE217LW:\
:rm=buster.math.its.rpi.edu:\
:rp=AE217LW:....

VC215LW|VCC_215_Private_LaserWriter:\
:sd=/usr/spool/VC215LW:\
:rm=netserv2.its.rpi.edu:\
:rp=sparclw:...
```

Figure 3: Beale printcap file Excerpt

On startup *etcprintcap* determines the current version of the printer database, and then compares it to the current version of the */etc/printcap* file. If things are up to date, it logs the fact that it checked in the **PRINTCAP\_LOG** table and exits. If, on the other hand, the printcap file is out of date, it selects the list of printers for that host (based on host groups), and then for each printer, selects the appropriate attributes, does any macro substitution needed, and writes the file. These steps are explained in more detail in the following sections.

### Version numbers

Each printcap file has a version number. This is essentially the maximum value of all the version numbers for each of the printers being included on the machine. Each printer entry in turn, has two version numbers, the maximum value of **Printers.When\_Inserted**, **Printers.When\_Updated** and **Printers.When\_Marked\_For\_Delete** which gives the printer version number, and the **Printers.When\_Attributes\_Updated** value which is updated by the *printmaster* program when a matching attribute is changed.

When the printcap file is generated, the newly determined version number is written in the header of the */etc/printcap* file (like in figures 3 and 4). The file version number is determined by reading the first few lines of the file and searching for the string "database version" and taking the number after it as the file version number.

### Host Groups

One of the requirements of the printcap configuration system, is the ability to have some printers only appear in the printcap files of particular host, or group of hosts. These special case printers

ranged from test printers that we were evaluating, to private dot matrix printers attached to someone's PC, to the printer that has the payroll check forms mounted.

---

```

General_Printcap
  All_Rcs_Hosts
    Engineering_Lab_1
      ELab1-01.rpi.edu
      ELab1-02.rpi.edu
      ...
    Engineering_Lab_2
      ...

Faculty_Hosts
  Math_Faculty_WS
    beale.math.rpi.edu
    buster.math.rpi.edu
  ...

```

---

Figure 5: Host Group Tree

We had already done some work with host groupings for generating usage statistics[3], so we decided to build on this existing work. The key to making this work effectively was to allow a group to be a member of another group. In this way, we could define a group, *General\_Printcap*, which most of the printers have set as the **Printers.Spool\_From** entry. This group formed the top of the host tree as shown in Figure 5. We can see that the machine *ELab1-01.rpi.edu* is a member of host group *Engineering\_Lab\_1* which in turn is a member of *All\_Rcs\_Hosts* which is a member of *General\_Printcap*. This way, when a new machine comes online, we just add it to the appropriate host group, and gets membership in the other appropriate groups.

---

```

#
# System printcap file for buster.math.rpi.edu.
# Generated by SIMON on 17-MAY-94 from
# printer database version 6358435
#
AE325LW|Amos_Eaton_325_Private_LaserWriter:\
:sd=/usr/spool/AE325LW:\
:rm=beale.math.rpi.edu:\
:rp=AE325LW:....

AE217LW|Amos_Eaton_217_Private_LaserWriter:\
:lp=/dev/ttya:\
:sd=/usr/spool/AE217LW:....

VC215LW|VCC_215_Private_LaserWriter\
:sd=/usr/spool/VC215LW:\
:rm=netserv2.its.rpi.edu:\
:rp=sparclw:...

```

Figure 4: Buster printcap file excerpt

Printers can of course, have a different group set in the **Printers.Spool\_From** column. If, for example, the printer AE217LW had **Spool\_From** set to *Math\_Faculty\_WS*, the entry for it would only appear on the machines in the Math department. A more detailed explanation of how to do nested groups is included at the end of the paper.

### String Substitution

Another key part of the printcap file generation, is being able to specify generic attributes, that then are filled in with the specific information for the current printer and host. As we can see going from the attributes in Figure 2, to the printcap file in Figure 3, the string "{printer\_name}" is replaced with the appropriate **Printers.Shortname** value.

We have a simple set of macro routines<sup>7</sup> that allow you to load values for a number of keywords, and then give it a line of text, and it will replace any {keyword} pattern, for the corresponding value. These keywords are updated for each printer as it is processed. We currently define the following keywords:

**Printer\_Name** which is the lowercase version of **Printers.Shortname**.

**printer\_type** which is the **Printers.Printer\_Type** value.

**connection\_type** which is the **Printers.Connection\_Type** value.

**spooler\_name** which is the actual hostname that corresponds to **Printers.Spooler**.

Since these macro routines are frequently used with Oracle database, we have added a *load\_row* sub-routine that loads the current row from the database, using the column names as the keywords. In the printcap example, all of the keywords except **Spooler\_Name** are loaded via the *load\_row* routine.

### Installing a new file

Since the *etcprintcap* program is intended to be run automatically in the early hours of the morning, we need to be very careful to fail in a safe way. To this end, we actually build the new printcap file in */etc/printcap.new*, and check the return codes from *fopen* and *fclose*, and only if everything checks out ok, we then rename */etc/printcap* to */etc/printcap.old* and then */etc/printcap.new* to */etc/printcap*. This does leave a very small window between the two renames where an ill timed system crash could leave a machine without a printcap file, but so far, we have not had any problems. By working to the side, we can also trap things like file system full and other errors.

<sup>7</sup>These are part of the Sandbox routines. See Availability for details.

The other part of installing the printcap file, is the creation of the spool and log directories. During the generation of attributes, each attribute line is passed to a routine called *check\_for\_files* that checks to see if the attribute is specifying a filename. If it is, it checks to see if the file (and parent directory) exist, and if not, create them with appropriate owner and permissions. This is not the prettiest solution, but it seems to work quite well. This would need to be expanded to handle a SysV printing system of course.

### PRINTCAP\_STATUS

Since the *etcprintcap* program records whenever it checks or updates a printcap file in the **Printcap\_Log** table, we also wrote a program, *printcap\_status*, that reads all the entries in the table and prints out any that have not reported in in the past 24 hours. It also flags hosts that have recorded a database version of 0. This has been useful in finding hosts that have had problems with the nightly updates and need intervention on the part of a system administrator.

### PRINTCAP\_LOG Table definition

**domain\_id number** The Simon.Dns.Domains.Domain\_Id of the host that this record is for.

**pcap\_version number** The version number of the printcap file from this host.

**last\_check date(7)** The date on when this host last checked the version of the printcap file.

**last\_update date(7)** The date on when this host last updated the printcap file.

### PRINTMASTER

The key to making all of this usable, is the *printmaster* program. This is an interactive program that assists you in manipulating printer definitions and printer attribute definitions. For the most part, the commands give you a way to display and update different columns in the **Printers** and **Printer\_Attr** tables, automatically converting from the internal formats<sup>8</sup> to a more human version as needed. It also provides some simple searching and selection functions to simplify locating a particular printer or attribute. The following top level commands are defined, most of which are for manipulating entries in **Printers**.

**list** Lists all the printers in the database. This isn't actually used very much anymore, as we have around 150 printers currently defined.

**destroy** This destroys a printer entry. It prompts you for a printer name. The actual destruction is

<sup>8</sup>For example, **Printers.Spooler** is simply a number, which then has to be joined with the **Dns.Domains.Id** to actually produce a host name.

done by setting the **Printers**. When **Marked\_For\_Delete** field to **Transcount.Nextval**. If needed, we could implement an undelete command.

**detail** This lists relevant information about a specific printer, basically columns selected from the **Printers** table.

**entry** This will generate the entry as would appear in the printcap file on both the spooler machine, and some other remote machine. This is very handy for checking that you have all the desired attributes set, and is much quicker than generating a full copy of */etc/printcap*.

**comments** This allows you to set or clear a few lines of comments that will be included in the start of the printcap file.

**connection** Allows you to change the **Printers.Connection\_Type** of the printer.

**printer** Allows you to change the **Printers.Printer\_Type** of the printer.

**group** Allows you to change the host group that gets this printer in its */etc/printcap* file. This can also be a single host if desired. While *printmaster* doesn't currently support this option, setting this to (Null) would have the effect of removing the printer from all printcaps, yet keep the name reserved.

**shortname** Lets you change the shortname of a printer. I have no idea why I never implemented a "longname" command.

**spooler** Lets you change the machine that drives the printer. This machine must be in the Hostmaster tables.

**alias** Lets you assign another alias for the printer. This information is stored in the **Printer\_Alias** table. This is a small table with basically just the **Printer.Id** value and the text of the alias.

**create** Allows you to define a new printer. If you are adding a new type of printer, you will need to add attributes as well.

**attributes** Drops you into the attributes subcommand mode, described below.

**touch** Updates the **Printers.When\_Updated** field. This will cause new printcap files to be generated. This is not normally needed, as any of the commands to change something, will also update these field automatically. It is handy for development and testing.

**types** Lists all the **Printer\_Types** and **Connection\_Types** currently defined in both **Printers** and **Printer\_Attr**, and the number of occurrences of each. This is often useful to remind you what types you have already defined.

**hosts** This lists all the hosts that have printers attached to them. The first part of the list are hosts with more than one printer, and this

includes the number of printers. The second part of the list, are all the hosts with just one printer.

**stop** Lets you out of the program.

To work on attributes, you select the **attributes** command and you then get the attributes menu. While in the attribute mode, the program defaults to the last connection type, printer type and rank that you specified. This can save a lot of typing when working on attributes.

**clear** Clears the default attributes from the previous command.

**clone** Copies a set of attributes from one printer type to a new printer type. We often get different printers that are all driven the same way, and the quickest way to load a new set of attributes, is to copy them from a similar printer and change the printer type on the fly.

**connection** Sets, clears or changes the **Printer\_Attr.Connection\_Type** on a particular attribute.

**create** Creates a new attribute. You are prompted for the different fields needed.

**delete** Deletes an attribute. As with printers, this is done by setting the **Printer\_Attr.When\_Marked\_For\_Delete** field. If needed, we could add an undelete command.

**entry** Changes the text of particular attribute.

**list** Lets you list attributes. You can specify any combination of printer name, printer type, connection type and rank to reduce the size of the list.

**printer** Sets, clears or changes the printer for this attribute. (Many attributes are not assigned to a specific printer.)

**ranks** Prints out each rank, the first 3 characters of one of the entries, and the number of attributes for that rank. This is very important to help keep the ranks and attributes matched up.

**stop** Returns you to the printer command menu.

The *printmaster* program automatically updates the **When\_Updated** column when a printer entry is changed, and updates the **When\_Attributes\_Updated** column for all matching printers when an attribute is updated. This helps *etcprintcap* detect when just a printer attribute has changed.

### Operational Status

The current version of the *printmaster* system has been in production use for about 2 months now (we were very glad to retire the old mainframe based system) and we have been very pleased with the results. Before we had automated the */etc/printcap* generation, adding a new printer was a hassle, often taking several hours to coax the old program to run, and we always had a number of machines with

printcap files that were months out of date. This was such a problem, that we really couldn't afford to offer a private printer service on faculty workstations. Even when we would install one, it generally took several hours of staff time to coordinate it, and often the change would be made by hand, and sometime later on, the faculty member would want access to some new printer, and the definition for their private printer would get overwritten. With the current system, I get a request with the required info, I spend 2 minutes with the *printmaster* program, and by the next morning, not only does the faculty workstation have the printer definition, all the rest of the campus hosts have it as well.

Many of our printers are connected via AppleTalk, and we drive them via CAP. Given our network layout, the choice of what print spooler machine to use to drive a given printer was often pretty arbitrary, and in the past, often depended on which spooler had a printcap file we could regenerate at the time. Now it is trivial to put the printer on any spooler, and moving a printer between spoolers is a single command, and waiting to the next day. **Note:** This can be a problem if the spoolers get updated in the wrong order. For moving active printers, you should be sure the update the new spooler before the old spooler, otherwise you may get jobs bouncing between machines.

### Future Directions

The printcap project has laid some groundwork for some other sys admin automation projects. Given the host group support, and some modifications of the techniques used to extract the lines for the printcap file itself, we plan on implementing a system to generate other system configuration files, such as crontab, syslog.conf, etc. This will be very useful as these files are generally the same for all of the systems, except for a few systems that need one line added or changed.

The *printmaster* program itself can use some more safety checks. Right now, it is possible for someone to mess things up a bit. An example of this are the ranks that are used in the **Printer\_Attr** table. Each of the two character capability codes as described in *printcap*(5) should each have it's own, consistent rank. This is only enforced by convention now, but should be enforced by the software. We don't have a lot of policy built into the program. While this may be a good thing in some respects, it does require a higher level of understanding and training on the part of folks who use it.

There are also some things that are not currently being used, such as the per page charge. I expect that as we rewrite the printer accounting system, these will start being used. Right now we have two lists of printers, the one in **Printers** and the one used by the current printer account system. That is one list too many in my opinion. We also will

eventually plug into the **Systems** table, but that is mostly for our own record keeping and not really related to printcap generation. Once thing that will give us however, is a standard Simon format location for each printer, which would be a prerequisite to providing a printer location service (*Where is the closest printer to this workstation?*).

### System V support

While it is likely that RPI will continue to use the Berkeley lpd/lpr/lpq suite of programs for some time to come (the changes we have made to it could fill another paper....) It would be close to trivial to modify Printmaster to generate /etc/qconfig and other SysV files. This would require changes to the code that generates the headers for each printer stanza (and in fact, it would make sense to move most of this to **Printer\_Attr** anyway), and the *check\_for\_files* would need to be updated.

Supporting both Berkeley and SysV at the same time would be a bit more work. You would want to add another field to **Printer\_Attr** called something like **SpoolSys\_Type** and then set that to **BSD** or **SYSV** for each attribute. What might be even simpler, is to define that BSD systems use ranks 1-9999, SysV uses 10000-19999, and so on. This would limit the changes to just the *etcprintcap* program. Not counting time required to double enter all of the attributes, the program changes would be on the order of a few hours.

### POSIX - Palladium

I have no direct experience with either Palladium[4][5] or POSIX P1003.7.1/D6[6], although I have read some papers on Palladium and one version of the POSIX draft. All sources concentrated on the actual printing process, and did not say much about managing the configuration. It appears that print servers still have printcap files, which could certainly benefit from this project. The clients get information from the Hesiod server, and it may be possible to provide the input data to Hesiod from this as well. However, this is simply speculation on my part.

### Installation Hints

As with any of the Simon modules, you need to obtain the Sandbox and Rsql modules. We (RPI) developed an API that sits on top of the oracle call interface (OCI). This also includes a network portion so that the program can run on a different machine than the database server. These have also been ported to run on top of Sybase, I can put you in touch with those folks if needed.

If you are willing to give up the host grouping functionality, you could remote the host group check, and replace the **Printers.Spooler** with the actual spooler name. This would allow you to avoid

installing the Simon Hostmaster<sup>9</sup> module, although you are exposed to problems with host name changes. A second option would be to install just the **Dns\_Domains** table and populate it with hostnames of hosts that will drive printers or will actually be running *etcprintcap*. Basically, this would be like installing Simon Hostmaster, but not actually populating most of the tables. We actually ran this way with the earlier version of printmaster.

Given at least the name part of Simon Hostmaster, the host group support would be trivial to install. It consists of two tables and a program to maintain the group memberships. (We also use host group memberships to generate host access control lists */etc/hosts.lpd* and */etc/netgroup*.) If you find that you also need to manage some part of your host name space, generate RR files for bind, etc, you may want to consider using more of Simon Hostmaster.

### Grouping in Oracle

As discussed earlier, one very useful module for printing support, is the host group work. A lot of the system management problems become easier to handle when you can break up machines into logical groups, and then join those groups together into other groups and so on (see Figure 5). While in this example we are doing groups of hosts, we use these same techniques for managing groups of unix ids (*/etc/group*) and groups of mail aliases (*/etc/aliases*).

We used two new tables for host groups. The first, **Host\_Groups**, is used to hold actual group name and has the following columns of interest:

**group\_id** *number* A Simon.People.Nextval that uniquely identifies this host group. This is a join column, and is based in this table.

**group\_name** *char(64)* The name of the host group. The host group name space is a flat space, with no hierarchy provided by Simon. In use, system admins may wish to follow a naming convention.

The second table is **Host\_Group\_Members** and is used to hold the membership of groups. The fields that we are interest in here are:

**member\_id** *number* The **Dns\_Domains.Domain\_id** or the **host\_groups.group\_id** of this entry. Since host ids and group ids are both drawn from the same sequence, they are globally unique.

**group\_id** *number* The **host\_groups.group\_id** of this entry.

When a group is created, it is assigned a **Group\_Id**. For each member in the group, a row is inserted into the **Host\_Group\_Members** table with the

**Group\_Id** and appropriate **Member\_Id**. For a normal host entry, this is the **Dns\_Domains.Domain\_id**, but for a group member, this is the **Group\_Id** of the group that is to be a member of the current group. Since both the **Group\_Ids** and the **Domain\_Ids** are drawn from the same oracle sequence, there are no collisions between the two sets of ids, and a host and a group can never have the same ID number. This makes things very nice when working with groups of groups.

To select the list of printers that we can should include on a given host, we want to find all printers where the **Spool\_From** is either the **Domain\_Id** of the current host, or the **Group\_Id** of a group that contains the current host. To do this, we execute the following bit of SQL code.

```
Select id,shortname,spooler....
from Simon.Printers
where spool_from in
  (Select distinct group_id
   from simon.host_group_members
   connect by prior
     group_id = member_id
   start with member_id=$HOSTID
  union select $HOSTID from dual)
```

The clause *where {ITEM} in {LIST}* matches all rows where the value in column {ITEM} is found in the list specified by {LIST}. In this case, we are generating a {LIST} of all the **Group\_Ids** that our host is a member of, along with the **Host\_Id** of the current host. This combined list is used to select our set of printers. Since we are working entirely with **Host\_Ids** and **Group\_Ids** here, we don't need the **Host\_Groups** at generation time. However, it is used by the *printmaster* program to set **Printers.Spool\_From**.

### Availability

All the source code for the Printmaster suite of programs, as well as table definitions, source code and additional reference material for the Hostmaster and host group modules are available for anonymous FTP from ftp.rpi.edu. See the file */pub/its-release/Simon.Info* for details on where to find everything. Some papers and presentations that discuss other parts of the Simon project are available in */pub/its-papers*.

If you have AFS available, you can browse many parts of the Simon tree. Look in */afs/rpi.edu/campus/rpi/simon/printmaster* for this project, and */afs/rpi.edu/campus/rpi/{sql,sandbox,netjack}* for other related parts. The anonymous ftp tree is also available in */afs/rpi.edu/campus/rpi/anon-ftp/1.0/common*.

If you just want to poke around, some information is available via <http://www.rpi.edu/~finkej/Simon.html>.

<sup>9</sup>Simon Hostmaster is another Simon module that manages all the data that goes into the RR files used for Bind and the */etc/hosts* file. It is a prerequisite for all of the Simon host configuration management modules. It is also freely available from ftp.rpi.edu.

### Author Information

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and communications programming with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past 3 years. Reach him via USMail at RPI; VCC 315; 1108th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu.

### References

- [1] Hein, T., & Nemeth, E. "Topics in System Administration I." *Tutorial Notes*, USENIX LISA VII 1993.
- [2] Finke, J. "Simon System Management: Hostmaster and Beyond" *Proc. Community Workshop '93*, Simon Fraser University, June 1993.
- [3] Finke, J. "Monitoring Usage of Workstations with a Relational Database" *Proc.*, USENIX LISA VIII 1994.
- [4] Handspicker, B., Hart, R., & Roman, M. "The Athena Palladium Print System" *Publisher Unknown* 9 Dec 1988.
- [5] Peisach, E. "Distributed Printing Within the Athena Environment" *Athena Technical Conference* Copy of presentation slides, date unknown, estimate 1990.
- [6] "IEEE std 1003.7.1 Draft System Administration Interface/Printing" *IEEE* October 1992.



# Highly Automated Low Personnel System Administration in a Wall Street Environment

*Harry Kaplan – Sanwa Financial Products Co., L.P.*

## ABSTRACT

Administering a small system running intensive financial applications with the demand for twenty four hour coverage is nearly impossible for a single person. Moreover, most solutions to automated system administration are not clever enough to pinpoint the problem and deliver the required action to the responsible person in a timely manner. Our system evolved by integrating small utilities which function as information gathering tools, notifying tools and action tools. Vital information to us means: the users on each system, load on each server, temperature in the computer room, remaining swap and UNIX file system space on each machine in the company, health of our market data ticker plant, and state of backup tapes. Notifying tools include electronic mail, electronic postit notes, a digitized dial-up voice messaging system, and alphanumeric pagers. Action tools include fairly standard UNIX utilities such as killing runaway application processes, removing core dumps, and, as such, require little further description.

## Introduction

This paper focuses upon routine information gathering and notifying tools which help a Wall Street systems administration department monitor critical resources on each workstation and server worldwide. No news is good news, so most of these routines run silently 24 hours a day until a problem is discovered. Aside from alerting the system administrators to take action, one unique feature of our environment is to "empower" the end user with their own tools for monitoring workstation resources.

## Motivation

The demands upon a system administrator in a Wall Street firm of some seventy-five employees worldwide are markedly different from that in other environments. Throughout various periods in the four year existence of the firm, I have been the sole administrator of one local and three foreign offices, designing everything from our wide area network to electronic post-it note facilities. Having a homogeneous environment of Sun Sparcstations running SunOS from the very start was a boon, but before we were able to grow the systems department to a staff of three, there was little time for complex programming tasks. Instead, the goal was to integrate as many standard, off the shelf products as we could purchase to save systems development time. Unfortunately, I discovered there is very little available to automate administration tasks, so we ended up creating tools which can serve as the thousand eyes and ears of our system. These tools help to identify trouble spots before they develop and even aid the user herself in identifying problem areas.

This paper presents a description of the utilities we have developed to help us maintain a near-zero downtime record on the trading floor, as well as survive the onslaught of component cave-in, memory saturation and even air conditioner failure in the computer room.

Small tools are built upon other simple tools. For example, here is the source code for our shell-level memory checking tool called "memory", which handles both Solaris environments and error conditions fairly painlessly. It also depends on a small script to determine the operating system level. There are numerous ways of doing the latter (checking for /vmunix for example) under SunOS, we chose a fairly arbitrary one, see Figures 1 and 2.

Another small script checks all ufs (4.2) file systems for disk space. Yet another tests that a given database server is active and healthy.

## Information Gathering Tools

### System Monitor

This is a cron job which runs each morning and mails the output to sysadmin, including snapshots of recent logs of the various things that run overnight. We found that the difficult part was to do a last log of all logins on all machines for the last two "interesting" days. A day is interesting if a new person has logged on to that machine in the interim. Figure 3 is the coding fragment that performs this trick.

This tool began as a single script which ran on every machine in the company. As the number of machines grew, this particular piece of mail became a chore to scroll through casually even on that first

strong cup of morning coffee. Over the years most of the subroutines have been broken out into separately labeled pieces of mail. The systems administrator must therefore browse through several dozen pieces of mail each morning to get a complete picture of the system status.

---

```
#!/bin/sh
if is_Solaris2
then
    MEM='/usr/sbin/swap -s'
else
    MEM='/usr/etc/pstat -s'
fi
OUT='echo $MEM |
    awk '{ print $(NF-1) }'
    | sed 's/....$//''
if [ "$OUT" -eq 0 ]
then
    echo 0
else
    echo $OUT
fi
```

Figure 1: Source for "memory"

---

```
#!/bin/sh
case `uname -r` in
    5.1|5.2|5.3) exit 0 ;;
    *) ;;
esac
exit 1
```

Figure 2: Source for "is\_Solaris2"

---

```
rsh $host last |
awk '{
    lastday = day
    day = substr($0,37,3)
    if (day != lastday)
        past++
    if (past > 2)
        exit
    print }'
```

Figure 3: Fragment of "sys\_monitor" source

We researched some of the packages available to create separate mailboxes into which these reports could be sorted, but the subject lines are not yet uniform enough to permit an automated distinction between system monitor mail and other messages intended for the system administration staff. In any case, speed reading mail messages becomes second nature for the well-seasoned system administrator, and we would not want to eliminate this vital skill from the growing portfolio of odd talents required for the job.

These system monitor scripts which collect information generally operate in a passive mode: the output must be browsed in order to be useful. This is fine for certain types of information. Recently,

however, we have added another layer of reporting which actively links certain log monitors (such as the database server dumps) to notification routines which search for errors of various kinds. One interesting error occurs when routines which are supposed to complete do not. A new procedure we have implemented watches the dozen or so databases as they are checked and dumped. If the database checking routine hangs or the dump does not complete, an appropriate notification routine is invoked. We have learned that in this specific case, trade entry may be jeopardized the following morning, so we would "like" to be woken out of bed by a buzzing pager when this occurs.

### Up Watcher

Once an hour, every machine on our network at each branch office is combed by the Up Watcher, and problems or potential problems are reported via pager or email. A remarkable amount of applications or systems level errors can be caught just by monitoring disk space and swap space events. Three times a day it is run in verbose mode which reports that everything is ok (along with computer room temperature) even if no problems are detected. This helps to insure that we know if the pager interface has crashed, or if our paging message vendor has suddenly closed shop and gone out of business.

Figure 4 is an outline of its logic.

---

```
For each machine on the network
  If it is not known to be down
    [ no entry in sundown ]
  If it is alive
    [ can ping and run simple command ]
    Check swap space
      > MIN_SWAP (10 MB)
    Check each 4.2 file system
      < MAX_SPACE (92%)
    Check user load
      < MAX_UPTIME (8)
  For each of the dozen local
  database servers
    Check that each database is
    up and running
  For each of the (two) foreign sites
  on high speed IP link:
    Check that the link is up
    (ping a remote machine)
    Check that the foreign database
    servers are up
  For each remote uucp site,
    Check to see if last successful
    connection > 1 hour
    Check to see if > MAX_WAIT (10 pieces)
    of outstanding mail are waiting
```

Figure 4: Logic schema of "up watch"

---

This system works well for us, except that you tend to get an annoying hourly message when the

cleaning person has knocked out the ethernet plug at 8 p.m. on a certain workstation on the trading floor. To this end, we created the /usr/local/sundown directory. By touching a file name corresponding to the machine that has been downed, an annoying hourly message that the machine is down will be suppressed.

The sundown scheme is also useful during crunch time on the development server, where we do not wish to know each hour exactly how perilously close to 110% the developer partitions have become, or what new exponent of user load has been reached during heavy compiles.

One interesting feature of the Up Watcher is that users are alerted and can take actions themselves. If Up Watch determines that the machine is low on memory the following electronic message is sent to the user:

*Your workstation \$i is precariously low on memory. You have only \$DOWNT0 Megabytes of Virtual Memory Remaining. This may adversely affect any applications currently running. Please save and quit any LARGE applications you are not using. Thank you. This message was generated automatically. A system administrator has been alerted.*

If the workstation is overloaded, the following message is sent to the user:

*Your workstation \$i is overloaded. There are currently an average of \$LOAD process waiting for cpu time. One of them may be sick and need attention. Please contact a system administrator if you believe you are not running any heavy applications on this machine. Thank you. This message was generated automatically.*

#### Feed Watcher

Similar to the Up Watcher, we run a feed watch three times a day at each office site where we are running our own in-house ticker plant to provide market data to the trading floor. It alerts us if any of the data feed provider machines have gone down or are not emitting any data. We even sample the type of data emitted to make sure that it is not garbage, and that it has been updating if the market is still open. Because we have distributed our market data feed system over many different server machines, Feed Watch also tells us which machine to log onto and which process to kill in order to restart a problematic market data daemon. It is often run manually if some problem in market data comes to the attention of the system administration staff, and helps to pinpoint exactly which component of the market data system is malfunctioning.

#### Adm Log Watcher

An adm log watcher on a designated workstation processes a tail -f on /var/adm/messages, and will alert us if an NFS server goes down for example, with the output

```
Apr 21 21:04:07 yukiguni vmunix: NFS
server kendo not responding still trying.
```

In this case, an immediate trip to the office after hours is advised.

#### Wiztemp

Wiztemp is a digital thermometer we purchased[1], and its daemon was easily customized to interface with our pager system. The temperature is always appended to any error messages output by the Up Watcher, so that we are warned of any impending disaster due to a rise in temperature in the computer room. In London we even have a digital thermometer on the trading floor, because some of our vital equipment was at risk when the air conditioning was shut off at night.

#### Stand and Stare

This is not an automated information gathering utility per se, although at times it appears to be so. This particular monitor occurs at various random points in the day when a user will stand over near the systems administration area and stare off into space as though in a trance. After some coaxing, it can usually be determined that a workstation has "frozen" up in a manner devastating to their normal business demeanor. Often a large application has been core dumping, and by the time the "stand and stare" monitor has sounded, the situation is back to normal. In the case where this occurs often, the user is often able to skip the trance state and announce the problem with little or no coaxing from the systems personnel.

#### Notifying Tools

##### Electronic Mail

The backbone of any system diagnostics will always be electronic mail. We adhere to the general UNIX philosophy that if things are good, we don't want to hear about it. Unfortunately, some of our diagnostic jobs are not smart enough to tell what is bad in a given log, and we are inundated with over 100 system-related e-mail messages a day. As mentioned above, retrofitting all of these messages with uniform and intelligent subject lines which announce GOOD or BAD results would ease the burden immensely.

Some logs and diagnostic routines are written only to files, and not sent through the mail, or are sent through the mail needlessly. For example, each day we run a search to find every file on the system which has been modified in the last 24 hours. This output is stored in hierarchical directories so that it is easy to figure out which tapes to use to restore files at a later date. The output is still sent to us via electronic mail, however. I actually find it profitable to skim through the contents of this mail message now and then because it alerts us as to areas in the company growing rapidly and possibly in need of additional disk space soon.

### SFP Postit Notes

Early on, we wrote a simple Open Look widget to deliver one-line postit notes to an end-user workstation. The message appears in a yellow rectangle with a date at the top, along with its sender and destination and a single QUIT button. To send SFP postit notes, you simply mail to alias "postit" and the user(s) you wish to send to are entered on the subject line. The postit program is then triggered through the standard mechanism of a mail alias pipe.

Over the years, various enhancements have been added to the postit system. If the user is not logged on at any workstation, or her X-windows security disallows posting one of these widgets, the sender is informed through electronic mail that it was unable to post for one of these reasons. All postit messages are also relayed through regular mail as well, so they may be picked up remotely. Few systems monitoring routines actually use the postit notification mechanism, because serious problems tend to occur when one is away from the workstation, or when the workstation is not operable in the first place. Notifications for failure of minor systems procedures like a failure in the daily dump tape management system can, however, be brought to the attention of the relevant party through the postit note mechanism.

### SFP Alphanumeric Pager Interface

The pager alias works identically to that of postit, except that in this case a group of programs actively take hold of a dedicated modem and deliver the mail message directly to our pager vendor's computer interface[2]. Our paging filter converts all newlines to spaces, and will create continuation chains if the message is longer than the eighty characters allowed per pager screen. The user id of the sender is prepended in parenthesis to the message for convenience. A "plus" indicates that the message will continue over more than one pager screen

The pager interface is used extensively to convey error messages which are flagged by the Up Watcher and Feed Watcher programs. Typical pager messages will look like Figure 5.

```
(Pager) zazen /data 96% temp is 64F
(dragon!pager) UV: All machines are up
and running [ Hong Kong office ]
(Pager) kaoru is down temp is 65F
(Pager) UK DB is DOWN 60.8F
(Ha Kaplan)+ Cannot get ahold of our
service vendor, typical, left a
(cont'd) message with mbar to reboot
tea tomorrow
```

Figure 5: Typical Pager Messages

The pager notification mechanism is vital to alerting us of system problems 24 hours a day. All systems administrators as well as our office manager and the developers in charge of the data feeds are required to carry their pagers with them at all times. During especially critical work periods one often sleeps with the pager next to one's pillow. I am capable of sleeping through my pager buzzing, but it wakes the dog who in turn wakes me. Sound sleepers are advised to find some suitable proxy method of responding to system emergencies.

### SFP Computerfone

The computerfone[3] interface provides many diagnostic services from any touch tone phone. This tool consists of the vendor-supplied hardware, a shell script and directories of digitized voice prompts and responses. Of course, with this system, the system response to a query must be fully anticipated, because the shell script uses a simple word matching algorithm to map error messages to enunciation. It is prudent to record the phrase "unanticipated data" as a default when no phrases are matched, rather than falling silent.

Using the SFP Computerfone interface, a trader in our Hong Kong office is able to check the health of a deal database in New York if a problem is suspected. More frequently, the computerfone is used to check the temperature in the computer room from an ordinary telephone on weekends during summer air conditioner inconstancies. By working through the touch-tone menu system we created, you can access the functions shown in Figure 5.

```
Leave audiotext e-mail for a sysadmin
Page a sysadmin
```

Check system functions:

- 1) Check to see when last e-mail went out to foreign offices
- 2) Check temperature in the New York computer room
- 3) Check to see if all database servers are up and healthy
- 4) Check to see if all T1 links are up and running
- 5) Check health of selected critical machines on the New York network as well as important machines in the foreign offices

Figure 6: SFP Computerfone Menu

Of course, a security code must be entered before the computerphone will provide data to an outside party. Care must be taken that no vital company information should be available through any dialup method, including voice.

## User Front Ends

### SFP MemoryTool

Putting system admin power in the hands of the end user is an idea which on the surface may seem dangerous or undesirable, but in our environment has saved many hours of systems department time. We designed the SFP MemoryTool to be a simple and effective way for the end user to monitor their own memory usage, to advise us if they feel they need more physical memory added to their system, or if they are dangerously close to exceeding the available swap space.

MemoryTool is an Open Look widget which looks like a thermometer when open. The top of the thermometer indicates 100% of the available swap space, and the physical memory limit is scaled automatically to the proportion of RAM to total swap. If the sum of all memory currently in use (psstat -s) is within physical RAM, the color of the thermometer is green. Once it exceeds this mark, as the mercury climbs up the memory tool it turns yellow. Within 10% of total available swap space it turns red. When iconified, the memory tool still changes color when crossing any of these thresholds. There is a button at the bottom of the thermometer which the user may click to get a detailed popup explaining all about how workstation memory works and how much her workstation is currently using. Figure 7 is an example explanation.

---

```

Your machine is loaded with 40M RAM
(FAST ACCESS) MEMORY
You have a total possible memory usage
(RAM AND SWAP DISK) of 117M
You are currently utilizing 12M actual
memory
This means that you are utilizing
30% of your RAM (FAST ACCESS)
memory and that you are currently at
10% of your total possible memory
capacity
  
```

---

Figure 6: SFP Memorytool Sample Explanation

Utilizing the Memorytool, users monitor the memory their applications consume and can determine on their own when it is necessary to eliminate one app before starting up another.

### Topmem

Some users on our system have never opened a shelltool except to run the "topmem" command. We teach them to identify and kill their own processes. Topmem is merely a filter we wrote which pipes the output of one invocation of the top[4] command and sorts it with regard to the resident application memory used by programs, avoiding displaying any processes using less than 200KB of memory. Programs which are heavily consuming memory resources are listed at the top, so the user can view

at a glance the application memory map on a particular workstation.

### To Do

We plan to isolate at least one workstation from both NIS and the automounter so that if any NIS or NFS servers are down it can tell us about it instead of freezing up and then telling us about it after the problem has been fixed. It would be nice to set up a queuing system for outgoing pages, but the collision rate has so far been less than the random message deletion rate of our pager vendor anyways, so this hasn't been a priority. It would also be nice to find a pager vendor who assigned various priority levels to system problems. It is annoying to be woken up only because a midnight compile triggered by a cron job has set off a disk space warning.

We would like to extend the Up Watcher in a few new directions. Monitoring ethernet load would be desirable, as would any slowdown in the speed of database transactions.

### Author Information

Harry Kaplan has designed UNIX systems on Wall Street for eight years, after having graduated from Berkeley before anything interesting was happening and having received a Ph.D. from Harvard in nothing relevant. He is currently the Systems and Technology Manager and Vice-President of Sanwa Financial Products, a limited partnership with The Sanwa Bank. He can be reached at 55 East 52nd Street, 26th Floor, New York City, New York 10055, or electronically at [harry@sanwaBGK.com](mailto:harry@sanwaBGK.com).

### End Notes

- [1] Wiztemp is available from Networks Wizards, PO Box 343, Menlo Park, CA 94026.
- [2] An alphanumeric pager interface program called "tipx" is available by anonymous ftp from [gatekeeper.dec.com: /b/usenet/comp.sources.misc/volume13](ftp://gatekeeper.dec.com:/b/usenet/comp.sources.misc/volume13). We did not use this program, but wrote our own.
- [3] Computerfone is available from Suncoast Systems, Pensacola, Florida.
- [4] The "top" program is available by anonymous ftp from [ecs.nwu.edu:/pub/top](ftp://ecs.nwu.edu:/pub/top).



# The Group Administration Shell and the GASH Network Computing Environment

*Jonathan Abbey* – The University of Texas at Austin

## ABSTRACT

Managing large scale UNIX networks so that users can use resources on multiple systems is traditionally performed using NIS, NFS, and DNS. Proper use of these tools requires exacting maintenance of multiple databases under a single point of control. This is acceptable for a few dozen machines, but with hundreds of machines spread across several administrative groups, some mechanism is needed to share administrative control over the master administrative files.

The Computer Science Division of the Applied Research Laboratories, The University of Texas at Austin (ARL:UT) has developed the Group Administration Shell (GASH) system, which gives different administrators permission to administer subsets of the laboratory NIS and DNS files. In addition to distributing administrative control in a secure fashion, GASH makes a large number of system administration tasks very simple, permitting technically untrained personnel to safely perform complex system administration tasks. The Network Computing Environment (NCE) built around GASH and standard UNIX tools such as NIS, DNS, and automounter provides a flexible and reliable way of completely abstracting the user's environment from the physical configuration of the network.

## Problem Statement

The proliferation of computer networking has outstripped the sophistication of the common mechanisms used to allow UNIX systems to share resources. It is not uncommon today for hundreds of UNIX systems to be linked together in a local area network. NIS and NFS [1] can allow those systems to share administrative information and file systems, but with such a large number of systems, centralized administration of the NIS databases becomes impractical. Local work-group administrators must either have root privileges on the NIS master server, with all the security risks which that entails, or they must submit their NIS change requests to a centralized administration group that is responsible for the maintenance of the NIS databases. Neither of these solutions is acceptable in an environment in which security and response time are critical.

There are other tasks that must be performed by a centralized administrative body which present an unacceptable bottleneck in administering systems in a shared network environment. For example, when a new system is added to the network environment, one or more IP addresses must be assigned. Detailed records of all system network information must be kept for network troubleshooting. There must be a simple and reliable way of keeping track of which systems and users have permission to access what NFS filesystems and network resources. Network filesystems must be mounted using standard naming conventions throughout the network

computing environment. E-Mail delivery must be coordinated to systems that cannot access a centralized UNIX mail spool filesystem.

All of these factors make large-scale systems administration extremely challenging. Even if it were acceptable to have single-point centralized administration of the NIS and DNS [2] databases, keeping the databases correct and consistent is non-trivial and time consuming. If a large number of part-time account administrators are to be able to safely modify the NIS databases, complex tasks such as renaming users or systems must be made simple. Name and numeric id conflicts within and between databases must be prevented.

## The GASH Network Computing Environment Solution

ARL:UT has gone a long way towards solving the problem of large scale distributed heterogeneous systems administration. By combining a set of carefully conceived administration conventions with a complex control and administration shell, NIS and DNS administration tasks can be divided among many part-time administrators. GASH performs a large number of complex tasks automatically, and constrains administrators to editing a restricted subset of the NIS and DNS maps, according to their administration privileges.

GASH controls a single, comprehensive NIS domain. All UNIX systems in the GASH NIS domain are part of the GASH NCE. In addition to

the basic user and group NIS maps, GASH maintains NIS maps to control the delivery of e-mail within the laboratory, individual and group e-mail aliases, automounter [3], and netgroups.

GASH maintains DNS tables as well as NIS maps, and is therefore involved in the administration of all systems in the laboratory, whether or not they participate in the NCE NIS regime. GASH keeps Ethernet and Internet records for all systems in the laboratory. All IP addresses in the laboratory are assigned by GASH, and all changes to IP addresses resulting from the physical movement of systems within the laboratory are handled by GASH.

The GASH NCE has been in conceptual development at ARL:UT for close to three years, and 40,000 lines of C code have been written over the past 18 months to implement the GASH control program. In operation for the last 9 months, GASH currently contains system information for 984 network devices (2011 total network interfaces) and NIS information for 614 users in 166 account groups. More than 45 individuals have been granted GASH administration privileges. These individuals range in expertise from seasoned systems managers to administrative and technical staff with little UNIX experience.

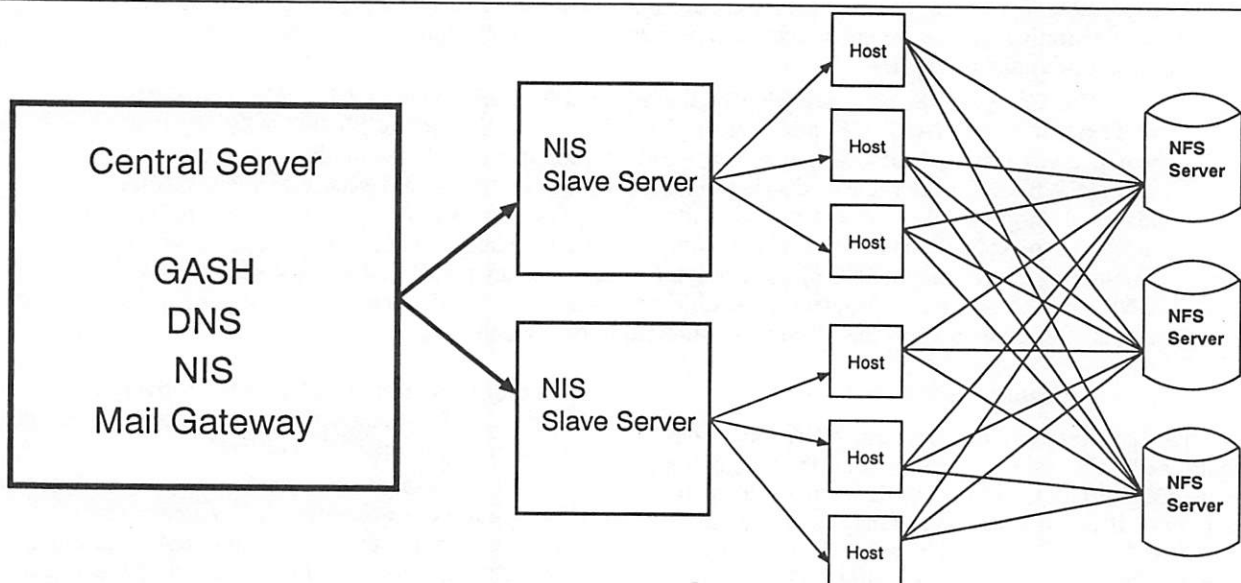


Figure 1: GASH NCE Physical Structure

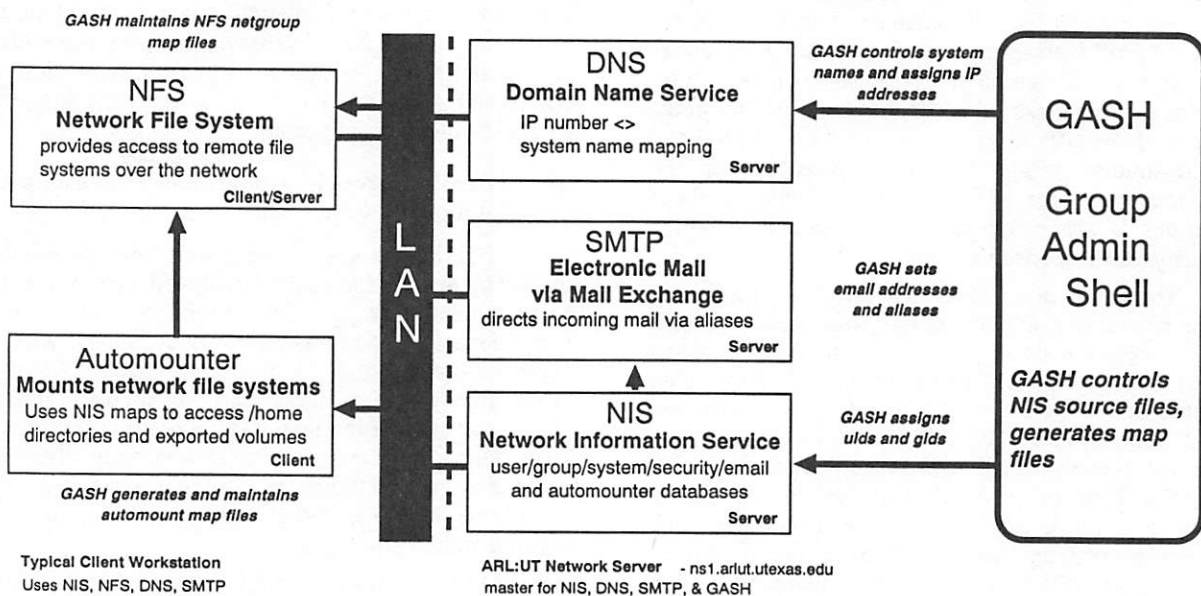


Figure 2: GASH NCE Logical Structure

### NCE Network Structure

The NCE consists of a central server that maintains NIS and DNS tables, multiple NIS and DNS slave servers, and a number of NCE hosts. Administrators run the GASH program on the central server to modify the NIS and DNS tables. Changes to the NIS and DNS tables are propagated to the slave servers, which in turn feed information to the NCE hosts. For reliability and performance, NIS slave servers are maintained on each subnet of our network.

The NCE supports replicated filesystem definitions for read-only NFS resources. Critical volumes (software archives and the like) may be redundantly served. NCE clients will access whichever file server is most convenient to them, according to the sophistication of their NFS client software.

Figure 1 illustrates the basic physical structure of the GASH Network Computing Environment just described. Figure 2 is an overall logical diagram of the NCE, detailing the ways in which GASH controls the NCE and its interrelated parts.

### GASH NCE Filesystem Conventions

The adoption of a number of lab-wide conventions has been critical to fully realize the benefits of the GASH NCE. The GASH NCE defines a uniform naming method for NFS filesystems that hides the actual physical location of the filesystem. All NCE access to NFS service goes through a level of

naming indirection that makes it possible to physically reconfigure NFS servers without disrupting access to logical volumes. Automounter is used for all NFS client access in order to insulate NCE clients from the vagaries of filesystem availability and to provide naming indirection. Automounter is configured on NCE clients using two primary automounter NIS maps, **auto.vol** and **auto.home.default**. GASH uses these NIS maps to centrally manage home directories and logical volume names for all NCE clients. A system without the ability to use automounter may directly participate in the NCE, but it must be locally configured to work with the NCE conventions. Any NCE client that does not use automounter will need to be manually reconfigured whenever any changes are made to network file servers.

NCE clients never refer to network filesystems by their machine name and path, but rather by a logical volume name assigned by GASH. At any time, a GASH administrator with appropriate privileges can change the association between a volume name and NFS filesystem. Such changes can be made to the network file servers without disrupting users that depend on access to particular volumes. All filesystems are automounted in **/v** using the **auto.vol** NIS automounter control map. For example, the volume **csdhome** may always be accessed through **/v/csdhome** by any NCE client, regardless of the physical location of the volume's filesystem.

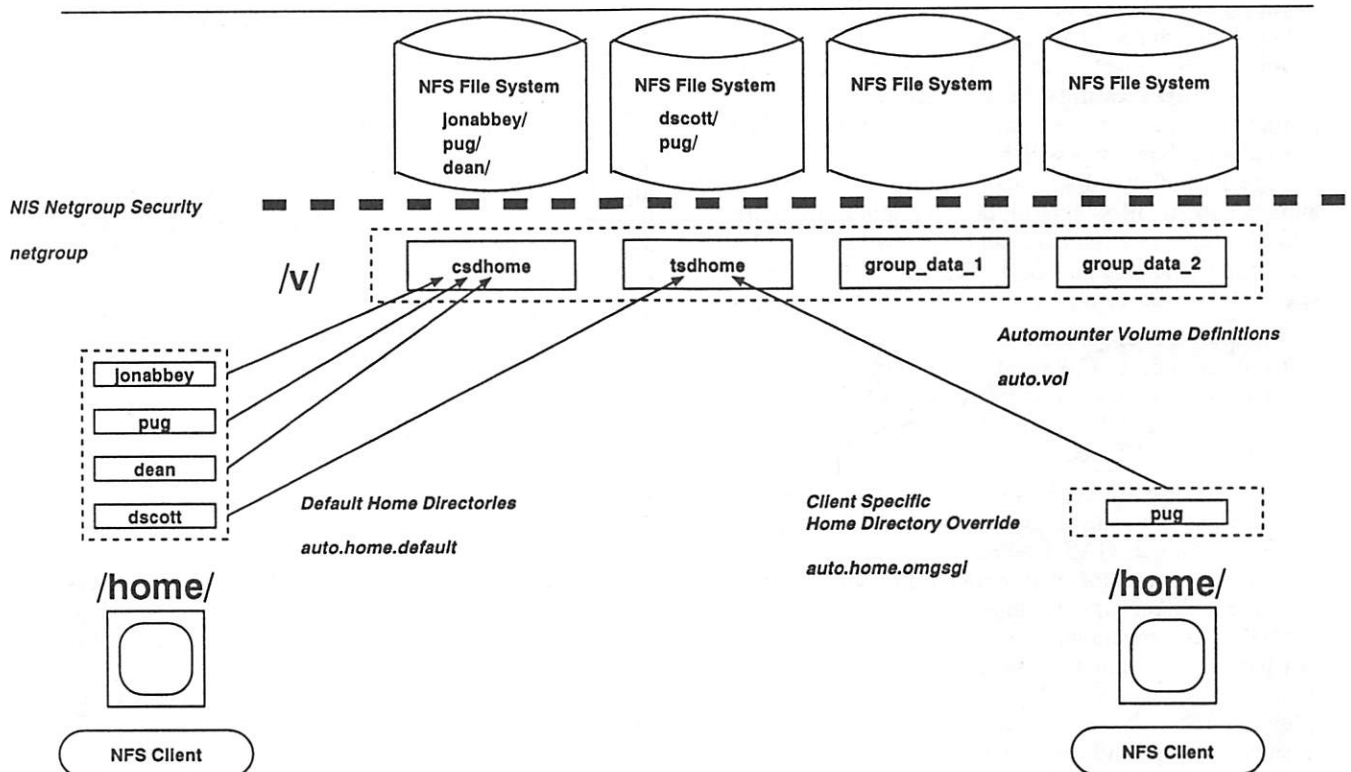


Figure 3: GASH Filesystem Management

All users' home directories are accessed through `/home/username` on all machines participating in the NCE. The GASH editor maintains automounter control files that map a user's home directory to a defined GASH volume. GASH administrators may create special control files to define alternate home directories for specific users on particular machines. A user may always access files on a particular volume by using the `/v` mechanism. If a user's default home directory is located on the volume `csdhome`, that user may always access that default home directory through `/v/csdhome/username`, even if he is logged into a machine using an automounter control file that specifies a different volume for his account's home directory.

NIS netgroups are used to control access to NFS filesystems. Each NFS file server in the NCE is configured so that permission to access exported filesystems is specified in terms of GASH managed NIS system netgroups. By using NIS netgroups in this fashion along with the automounter NIS maps, it becomes possible to perform all network access administration through GASH once an NFS server is properly configured. Figure 3 illustrates the way in which the GASH controlled automounter and NIS netgroups control the NFS network environment.

#### GASH Client and NIS Conventions

To fully work with the GASH NCE, all host systems must be configured to use the `passwd`, `group`, and `mail.aliases` NIS maps in addition to the automounter maps. In order to use the `passwd` and `group` NIS maps, the `/etc/passwd` and `/etc/group` files must be modified to specify NIS chaining with the '+' syntax [1, p.32]. The `mail.aliases` NIS map will be used automatically by a properly configured sendmail [4]. Users' `.cshrc` and `.login` files should be configured to work with the NCE environment, and the `/etc/shells` file should include all shells defined in the GASH `shell_paths` control file.

All GASH NCE uid's and gid's are at or above a fixed boundary (1000 at ARL:UT). All local `/etc/passwd` and `/etc/group` entries should have uid's and gid's below the boundary to avoid conflicts with NCE accounts.

In the GASH NCE, all UNIX account groups are six characters in length. The first three characters are a unique GASH administrator code indicating the administrator that created the group. The last three characters are unique to the group, and generally are mnemonic in nature. Netgroups and auxiliary automounter control files contain similar administrator codes. These administrator codes allow GASH to restrict editing access to these objects and provide an easy way to determine quickly which administrators are associated with a group, netgroup, or automounter control file.

#### Other NCE Components

One of the goals of the NCE is to provide users with a consistent working environment throughout the laboratory. Supporting a single home directory throughout the NCE goes a long way towards this goal, but does not solve some important problems.

The first problem is that users' `.cshrc` and `.login` files are usually not portable across different kinds of UNIX systems. This problem was dealt with by splitting the traditional home `.login` and `.cshrc` files into two parts, one system dependent and one system independent. System dependent initialization is performed by `/etc/.login` and `/etc/.cshrc`. These files set the default `PATH`, `MANPATH`, and `LD_LIBRARY_PATH` environment variables, as well as any other system specific environment or shell variables. An NCE user's home `.login` and `.cshrc` files reads the system dependent `/etc` files on startup. The home `.login` and `.cshrc` files contain sections that allow customizations targeted to specific types of systems, along with a generic customization section.

The second problem is that the difficulties of managing application software are magnified in a large network environment. The location of software should be transparent to users, and software packages should be easy to update or remove. To handle these issues, a sophisticated software management suite called `opt_depot` was developed. `opt_depot` was inspired by Carnegie Mellon's Depot software [5], and allows easy maintenance of a single central package archive for freely distributable and site licensed software. System administrators in the laboratory can configure their workstations to access these packages from the central package archive or from a local copy without any recompiling or re-configuring. The central package archive, if used, is accessed using the volume indirection conventions of the GASH NCE, giving us the flexibility to relocate or replicate the archive as needed. The `opt_depot` software integrates all packages into the `/opt` directory structure, regardless of package location.

In addition to handling NIS and DNS, the central server acts as the laboratory's primary mail server and gateway. The server exports a mail spool volume, `/v/mail`, that is mounted by NCE hosts. Any mail entering or leaving the laboratory is processed by the server in accordance with the NIS `mail.aliases` map maintained by GASH. Mail leaving the laboratory can have its headers rewritten to reflect a canonical laboratory e-mail address for the sending user. Mail entering the laboratory may be redirected to machines that do not take advantage of the central mail server's mail spool.

### GASH Command Tool

Much of the work in developing the GASH NCE was spent in the construction of the GASH command shell. GASH is a single-user, text-based interactive editing shell and DNS/NIS manager that provides task oriented (rather than file oriented) editing and browsing facilities. The GASH command shell is very configurable, and has many convenience and security features.

#### GASH Command Shell Commands

GASH contains 32 primary commands, grouped into eight sets of four commands. Each set includes a command to create, modify, inactivate, and list the appropriate data type. For instance, the user set includes `cua`, `mua`, `iua`, and `lua` to create, modify, inactivate, and list users. Sets exist for **users, groups, netgroups, systems, e-mail aliases, volumes, automounter control maps, and administrators.**

GASH commands are designed to allow the administrator to make changes to everything that relates to the subject at hand. While editing a user account, for instance, a GASH administrator may cause changes to be made to several groups, netgroups, and e-mail aliases. This tight binding between the GASH databases will be discussed in depth later in the paper. In general, the GASH administrator does not need to know anything more than what needs to be accomplished. GASH commands accept defaults, and the user can terminate any command all desired changes have been made.

The **create** commands will typically do everything necessary to get an object of the appropriate type ready for use. For instance, creating a user account with the `cua` command will prompt the administrator for a list of groups and netgroups to which the user account should be added, the volume on which to place the user's home directory, and the user's default e-mail address.

The **list** commands typically only show data items that the administrator has permission to edit. The exception to this is the list volume command (`lvn`), which shows all volumes defined in the NCE. Inactivated items may or may not be visible, depending on the particular command. The **list** commands accept command line parameters, which the **list** commands will try to intelligently match against the portion of each database entry that the user might wish to search.

The **inactivate** commands may remove an object immediately (systems, e-mail aliases, administrators, volumes), or may cause an object to be removed at a later time (users, groups, netgroups). Objects to be removed at a later time are immediately made unusable, and entries describing the actions to be taken to ultimately remove the objects are made in the GASH `pending_actions` file.

This delayed removal is necessary for many kinds of database entries. Users and groups must not be removed immediately, because doing so will cause confusion in the NCE file systems. By delaying final removal, GASH provides a consistent environment for local system administrators to do clean-up, and prevents new users or groups from being granted the newly available uid or gid too soon.

When an object is removed from the GASH databases, all references to it are cleanly removed. An object may generally not be inactivated or removed if this will cause any GASH database to become inconsistent. Administrators must have editing authority in order to inactivate an object. See the sections on the individual GASH databases and the `pending_actions` control file below for more information on object inactivation.

The **modify** commands allow administrators with editing authority to change a logical object. As with the **create** commands, modifying a single logical object may cause changes to all databases that reference the object. Modifying a user with the `mua` command, for instance, will prompt the user to specify groups and netgroups from which the user should be added or deleted as part of the editing process. Whenever GASH prompts for a modification to a list of objects, the administrator may enter a list of objects to be added or removed from the list. If an object that the administrator entered was on the list, it is removed. If the object was not on the list, it is added.

#### GASH Security and Administrator Privileges

The GASH command shell includes several features to preserve NCE security. GASH uses its own password file to control access to the command shell. The machine that GASH is running on only allows logins from users who have been granted GASH privileges. GASH has a ten minute time out feature, and will re-prompt for the administrator's password if it is left idle for more than five minutes. GASH will not display an administrator's privileges unless the administrator's password is correctly re-entered. All incorrect password attempts are logged, and mail is sent to the system administrators overseeing GASH.

GASH supports a **supergash** account that can take action on behalf of any group administrator. The password to this account is guarded at least as closely as the system root password. At ARL:UT, considerably fewer people know the GASH **supergash** password than know the Computer Center's root password.

Each GASH administrator is assigned a range of uid's and gid's for which the administrator has editing authority. In addition to uid and gid range permissions, each administrator has a unique three letter identification code and a set of privilege masks

to control access to groups, netgroups, automounter control files, and e-mail aliases not attached to a user account. The use of privilege masks allows a group of administrators to have overlapping privileges. An administrator might have an identification code of `omj`; another administrator, with an identification code of `omg`, might have an automounter control file mask of `om*`, allowing full access to automounter control files generated by either administrator.

GASH administrators do not necessarily need to have access to each of the eight GASH command sets. Permission to use each of the sets can be individually granted or denied to administrators.

#### Logging / E-Mail System

GASH has a sophisticated system for tracking modifications to the NCE databases. Each individual database modification may result in an event being generated. Each event is logged to a `gash_log` file, and may be logged to the system `syslog` file.

Many events result in e-mail being sent to the GASH administrator or administrators responsible, as well as any users or system managers who should be notified of the event. The `mail_control` control file contains a list of event codes, along with a list of administrators or users that should be notified whenever an event of that type occurs.

#### Pending Actions / Nightly Processing

Many system administration actions require an initial action followed by a clean-up action some time later. Removing a user, for instance, requires that the user's password and login shell be changed, but the user's GASH records need to be maintained until such time as all files associated with the user's uid can be removed. GASH has a facility for writing notes to itself for later action. These notes are written into the `pending_actions` control file. Whenever GASH makes a note in the `pending_actions` file of an action to be taken in the future, it also makes notes to send mail to the users affected by the action. Users will typically receive mail once a week until the action occurs, reminding them of the action to be performed.

Each night, GASH runs in an automated mode that performs any scheduled actions from the `pending_actions` file and does cross-database consistency checks. GASH compiles a nightly report that includes any inconsistencies detected, along with a list of any pending actions performed. This report is sent to a list of administrators specified in the `mail_control` file.

#### GASH Databases and Control Files

GASH manages a minimum of seven administration databases. These databases are listed in Table 1. The NIS and DNS tables generated from the GASH databases are listed in Table 2.

Filename	Purpose
<code>user_info</code>	Users
<code>aliases_info</code>	E-Mail
<code>auto.vol</code>	Volumes
<code>auto.home.*</code>	Home Directories
<code>group_info</code>	Groups
<code>hosts_info</code>	Systems
<code>netgroup</code>	NIS Netgroups

Table 1: GASH Databases

Map	GASH Source File
NIS <code>auto.vol</code>	<code>auto.vol</code>
NIS <code>auto.home.*</code>	<code>auto.home.*</code>
NIS <code>ethers</code>	<code>hosts_info</code>
NIS <code>group</code>	<code>group_info</code>
NIS <code>hosts</code>	<code>hosts_info</code>
NIS <code>mail.aliases</code>	<code>aliases_info</code>
NIS <code>netgroup</code>	<code>netgroup</code>
NIS <code>passwd</code>	<code>user_info</code>
DNS named tables	<code>hosts_info</code>

Table 2: NIS and DNS Tables Generated from GASH Files

The GASH databases are as similar as possible to the standard `/etc` files and NIS source files upon which they are modeled. The `netgroup` database is actually identical to the standard NIS `netgroup` source file. They are all line-oriented text files, and can generally be edited by hand with little trouble. The exception to this is the `hosts_info` file, which is quite complex, and formatted quite unlike any standard NIS or DNS source file.

GASH includes several perl scripts that are run by a makefile at the end of each GASH session in which databases have been changed. These scripts take the several GASH databases and transform them into the standard files expected by NIS and DNS. The makefile then takes whatever action (pushing NIS maps, updating DNS named tables) is appropriate to propagate the new information into the NCE.

One of the primary responsibilities of the GASH program (and the primary source of its complexity) is to maintain constraints and relationships within and among these databases. These constraints may be as simple as making sure that a user's account name in the `user_info` database is unique and less than 8 characters, or as complex as making sure that all relevant e-mail addresses in the `aliases_info` database are updated when a system in the `hosts_info` database is renamed. One of the best ways to get a good idea for how GASH works is to review in detail the databases that it manages. The next section will discuss the GASH databases and the relationships that GASH maintains between them.

Figure 4, below, shows the seven primary GASH databases along with the GASH administrator control file. The lines connecting the databases indicate cross-database references. The arrows indicate the direction in which automatic changes can propagate through the databases. Not all connections are shown.

#### The user\_info GASH Database

The **user\_info** database is one of the primary GASH databases. It contains an entry for each user granted a UNIX account in the NCE. Each entry is similar to that found in the traditional `/etc/passwd` file, with user account name, password, user id, default group id, a structured GCOS field providing user identification, the user's home directory (`/home/username`), and the user's login shell. In addition, the **user\_info** database contains the social security number for each user. This social security number is used by ARL:UT as a unique identifier to insure that NCE accounts are granted only to valid personnel, and that users are not given duplicate accounts.

Within the **user\_info** database, GASH guarantees that each user's name and id will be unique and will conform to the length and character set restrictions of standard UNIX. GASH assigns a free uid in the creating administrator's uid range when a new user is created. GASH always sets the home directory field to `/home/username`, in accordance with NCE convention. The automounter control maps (**auto.home.default** and any specialized control maps) determine where `/home/username` is ultimately located.

There are many relationships that GASH maintains between the **user\_info** database and the other GASH databases. A user's default group id is guaranteed to refer to a valid group in the **group\_info** database containing the user. The user is guaranteed to have an entry in the **aliases\_info**

database, and an entry in the **auto.home.default** database, in addition to at least one entry in the **group\_info** database. In addition, the user's name is guaranteed not to conflict with any alias names registered in the **aliases\_info** database, and is guaranteed to be changed in all databases managed by GASH if the user is renamed.

A user account may be inactivated, in which case the user's password will be set to `'*'`, and their shell will be set to `/bin/false`. When a user is inactivated, a record is placed in the **pending\_actions** file which will cause the user to be removed at a future time. Inactivated users have their e-mail address in the **aliases\_info** database retargeted to the administrator who inactivated them. Inactivated users may be reactivated by a GASH administrator using the `mua` command. A reactivated user account must have a new password, login shell, and (optionally) e-mail address assigned.

An active user account may instead be transferred to the control of a GASH administrator. When this happens, the user's GCOS information and social security number are changed to that of the GASH administrator. If the user account is registered as a GASH administrator account in the **admin\_info** control file, inactivating or transferring the user will remove GASH administration privileges from the user account.

#### The aliases\_info GASH Database

The **aliases\_info** database is the master e-mail addressing database for the NCE, associating simple names with fully qualified e-mail addresses. These names may correspond to users in the **user\_info** database, external users (individuals with an NCE address who do not have UNIX accounts and may in fact be receiving mail outside of the laboratory), or group e-mail accounts. Each of these three types of aliases have specific constraints associated with them that are maintained by GASH. Modification

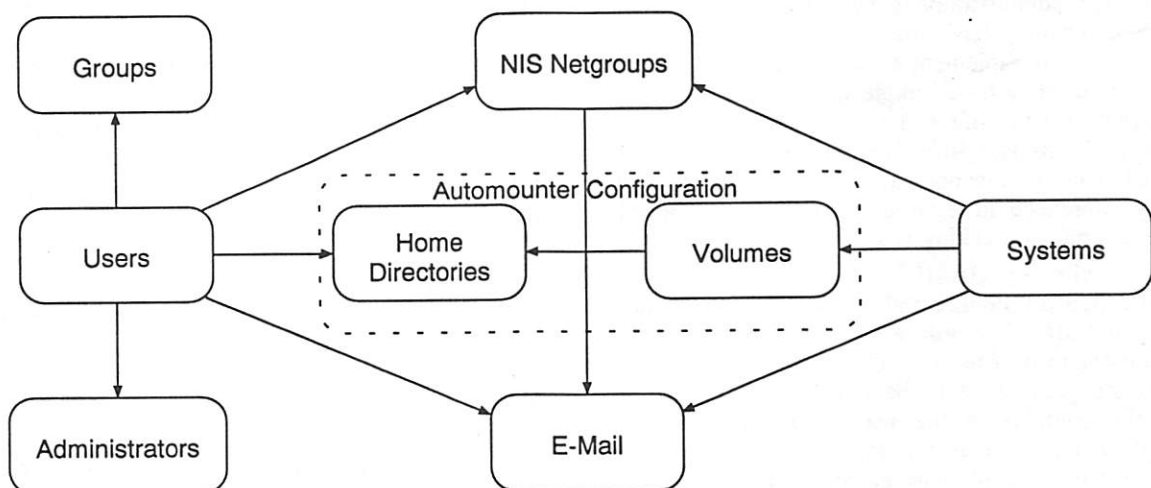


Figure 4: Logical Relationships Between GASH Databases.

privileges for NCE user aliases are controlled by uid range. Modification privileges for external user and group aliases are controlled by a three letter code indicating the identity of the administrator that created the alias.

User and external user alias records are identical, with the exception that external user alias records have an additional field to hold the administrator code. Each user and external user record consists of the canonical user name (identical to the NCE user name in the case of NCE user aliases), a set of alternate names, and the e-mail address or addresses to which the user's e-mail should be forwarded. One of the alternate names may be chosen to be the "signature" name, which will be placed in the "From:" header on all e-mail from the user which leaves the NCE.

In the following example (Figure 5), NCE user broccol has jonabbey as an alternate name. Mail addressed to this user in care of the NCE (e.g., jonabbey@arlut.utexas.edu or broccol@arlut.utexas.edu) will be redirected to broccol@arlut.utexas.edu. Mail messages from this user passing through the NCE mail server to destinations outside of the NCE may have their headers re-written, when using an appropriate version of sendmail [6] on the NCE mail gateway. If this happens, the user's account name is replaced with the signature alias (the first entry in the alternate names field, in this case jonabbey). Mail from broccol@cstdsun1.arlut.utexas.edu will be received as having come from jonabbey@arlut.utexas.edu.

---

```
broccol:jonabbey,
      broccol:broccol@arlut.utexas.edu
```

---

Figure 5: Example User Alias Record

---

Group e-mail alias records are essentially external user alias records without the alternate names field. The GASH program draws a distinction mainly for administrator convenience. All e-mail alias records may have multiple e-mail targets and may be used to implement e-mail groups. All e-mail alias records may have simple names in their target field, provided the simple names are registered elsewhere in the **aliases\_info** database. GASH converts any alternate (non-canonical user or external user) names referenced in a target field to the canonical name to simplify tracking target alias changes.

As with all GASH databases, a number of sophisticated constraints and relationships are maintained that affect the **aliases\_info** database. Within the **aliases\_info** database, all names and alternate names are guaranteed to be unique, and to cohere with the contents of the **user\_info** database. By default, whenever a user is renamed, the user's old name becomes an alternate name for the user's new name, preventing the possibility of dropped e-mail during the transition.

Simple name e-mail targets are guaranteed to exist elsewhere in the **aliases\_info** database. E-Mail alias loops may exist, but GASH does check to make sure that an alias does not target itself. If an e-mail alias is removed, all aliases that targeted that alias will have the alias removed from their target fields. Likewise, any time a user is renamed, all aliases that target that user will be updated with the new name.

Absolute e-mail targets (i.e., e-mail targets that include an '@') that refer to machines in the **hosts\_info** database will be kept synchronized with any major changes (such as system renaming or deletion) in the **hosts\_info** database.

GASH automatically maintains group e-mail aliases for all user netgroups in the **netgroup** database for administrative convenience. These aliases may not be edited directly; any changes to the corresponding user netgroup will be reflected in the alias.

### The auto.vol GASH Database

The **auto.vol** database contains a list of all NFS volumes defined in the NCE. Each record consists of a volume name and a list of one or more NFS filesystems (host and path) to which the volume name refers. Multiple NFS filesystem entries are typically used to list multiple interfaces on a single machine, but may also be used for replicated filesystems.

Volume names are guaranteed to be unique within the **auto.vol** database. GASH maintains relationships between the **auto.vol** database and the **hosts\_info** and **auto.home** databases. Volumes are guaranteed to refer to systems registered in the **hosts\_info** database. Systems may not be removed from the **hosts\_info** database if there are any volumes that solely reference the system to be removed. An administrator must remove the volume definition before removing the system or systems from the **hosts\_info** database. Likewise, a volume may not be removed from the **auto.vol** database if there are any entries in any of the **auto.home** databases that reference the volume to be removed.

In the GASH NCE, the **auto.vol** database is used to directly control the automounted **/v** directory. All volumes in the NCE may be accessed through **/v/volumename** on all systems with appropriate access privileges.

Volumes may only be created, modified, or destroyed by administrators who are listed as managers of one or more of a volume's component systems.

### The auto.home GASH Databases

Just as the **auto.vol** database is used to control the automounted **/v** directory in the NCE, the **auto.home** databases are used to control **/home**. Each **auto.home** database consists of a sequence of

entries binding an NCE user with a volume from **auto.vol**. Each user may appear only once in each **auto.home** database. Volumes are guaranteed to exist in the **auto.vol** database.

There is one special **auto.home** database, **auto.home.default**. All NCE users are guaranteed to have an entry in the **auto.home.default** database. Under normal conditions, a user will have an entry in **auto.home.default** only, and all machines that the user uses in the NCE will mount **/home/username** from the volume specified in **auto.home.default**. Occasionally, however, a user will need to have a different home directory on a particular machine or set of machines. In this case, the user will have an entry in an auxiliary **auto.home** database, and the machines in question will be configured to search through the auxiliary database before **auto.home.default**.

Auxiliary **auto.home** databases are given names of the form **auto.home.identifier**, where **identifier** is composed of an administrator code and a unique identifier. Administrators with appropriate administrator codes may add, change, or delete any entry in an auxiliary **auto.home** database. Administrators who do not have administrator code access to an auxiliary **auto.home** database may still add, change, or delete any entry referring to an NCE user under their uid range. All administrators have access to the **auto.home.default** database on this basis.

When the **auto.home** databases are turned into NIS maps, the volume name in each entry is replaced with the current definition of the volume. If any volumes are ever modified, GASH will regenerate all the **auto.home** NIS maps with the new filesystem information. In this way, the definition of volumes may be changed at any time and the change will be reflected almost immediately throughout the NCE.

#### The group\_info GASH Database

The **group\_info** database is essentially the standard UNIX **/etc/groups** file with a pair of extra fields added. The first additional field is an ARL:UT accounting code. The second additional field contains a description of the project or administrative group associated with the account group. The password field is not used for active groups in the NCE. GASH assigns a password of **zzzz** to active groups, and **\*\*inactivated\*\*** to inactivated groups.

Within the **group\_info** database, GASH guarantees that group names are unique and conform to the length and character set restrictions of standard UNIX. All gid's are guaranteed unique; new groups are created with a free gid from the creating administrator's gid range. Between GASH databases, GASH guarantees that all users referenced in the **group\_info** database actually exist, and will not

allow an administrator to remove a user from his default group without either changing the user's default group or inactivating the user.

Editing access for groups is restricted both by administrator gid range and by administrator control code. Administrators with permission to edit a particular user account may perform actions that will result in changes to all groups referencing that user, regardless of individual group permissions.

#### The hosts\_info GASH Database

The **hosts\_info** database keeps a complete list of all systems connected to the laboratory network using IP addressing. For each system, the **hosts\_info** database records the system's main name, any specific interface names or CNAME aliases, the room that the machine is located in, a list of authorized system managers for the machine, and all Internet and (optionally) Ethernet addresses associated with the machine. In addition, information is kept on the type of system, the manufacturer and model, and the operating system being run.

GASH guarantees that all system and host names are unique, and that all IP addresses are unique. GASH uses the **networks\_by\_room** and the **internet\_assign** control files to assign IP addresses that correspond to the type of system and to the networks that connect to the locale (room) containing the system. GASH guarantees that a system will not be removed if a volume is defined on that system. If a system is removed or renamed, appropriate action is taken in the **aliases\_info** database, the **netgroups** database, and the **auto.vol** database. If a system manager's user account is removed or renamed, this is likewise reflected in the **hosts\_info** database.

GASH supports moving a system from one room to another, and will carry over all Internet addresses that are valid in the new location. GASH supports multiple interfaces on systems, and can add or remove interfaces at any time. GASH has special support for SLIP users; if a system is of type SLIP, the first manager listed in the system manager field will be considered to be the name of the NCE user being granted SLIP privileges. The makefile that runs when GASH exits editing database files will use this information to build SLIP control files.

Permission to modify host entries is restricted to those administrators who are listed in the system manager field of a system. Only those administrators may remove or modify a system, or volumes that depend on such a system. Any valid GASH administrator with system management privileges may add new systems.

#### The netgroup GASH Database

The **netgroup** database contains NCE-wide definitions for user and system NIS netgroups. Each record contains a netgroup name along with a list of netgroup entries that belong to the netgroup. Each netgroup has a name of the form <3 character

code><identifier>{-u|-s}, with user netgroups ending in -u and system netgroups ending in -s.

Standard NIS netgroup entries are tripartite, containing a field for a user name, a system name, and an NIS domain name. Groups of these entries, along with references to other netgroups, make up a standard NIS netgroup. Netgroups are used to grant or deny permission to access a resource to a group of entities. In practice, there is no occasion to grant or deny access to a group that includes both users and systems [1, p. 167].

In GASH, netgroups are simplified; system netgroups can only contain system netgroup entries and references to other system netgroups, and user netgroups likewise. A GASH netgroup can thus be thought of as just a list of user names or system names. The NIS domain name field always contains the name of the NCE NIS domain name. This allows NFS servers to verify that client access requests are coming from systems within the NCE.

GASH guarantees that all users and systems referenced in GASH netgroups actually exist in the **user\_info** and **hosts\_info** databases. Any user or system renaming or deletion results in the appropriate entries in the **netgroup** database being modified or deleted, as appropriate. GASH guarantees as well that any references to other netgroups are valid, and will remove netgroup to netgroup references when a netgroup is removed. GASH maintains group e-mail aliases (without the trailing -u) for all user netgroups in the **netgroup** database as an administrative convenience.

GASH restricts access to netgroups to those administrators who have an appropriate netgroup code mask in the **admin\_info** control file. Systems and users may always be removed or renamed by an administrator with appropriate permissions, and the corresponding changes will be made throughout the **netgroup** database, regardless of individual netgroup permissions.

Filename	Purpose
admin_info	Administrator Privileges
mail_control	Diagnostic Mail Targets
networks_by_room	Room Connectivity
pending_actions	Future Actions
shell_paths	Default Shell Choices
internet_assignment	IP range control

Table 3: Major GASH Control Files

### GASH Control Files

In addition to the primary databases that GASH manages, there are a number of control files that GASH consults as it goes about its job. Table 3 contains a list of major GASH control files, along with their general purpose.

### admin\_info

The **admin\_info** database contains a list of NCE users who have been granted GASH administration privileges. Table 4 shows the fields present in the **admin\_info** database.

Field	Purpose
user name	NCE user name
password	Administration password
admin code	Unique administrator code
commands	Command set privileges
low uid	Lowest uid in authority range
high uid	Highest uid in authority range
low gid	Lowest gid in authority range
high gid	Highest gid in authority range
group mask	Group name privileges
user netgroup mask	User netgroup privileges
system netgroup mask	System netgroup privileges
email mask	Group, external user alias privileges
automounter mask	Automounter control files privileges
email address	Optional email address

Table 4: admin\_info Fields

The first two fields in Table 4, user name and password, are self-explanatory. The third field is a unique three letter code that is embedded in any group, automounter database, netgroup, or e-mail alias created by the administrator. The commands field is a bit field granting or denying the administrator permission to execute the various GASH command sets. The next four fields grant the administrator privileges to edit a subset of the user uid and group gid ranges. The next five fields, the mask fields, are masks to match against the previously mentioned embedded administration codes. These masks may be up to three characters in length, including an optional trailing wild card. The last field, the optional e-mail address field, is used for those administrators who want e-mail resulting from their administrative actions to go to an address different from their personal address. This is often useful for automatically notifying a group of administrators of one's actions.

As with the GASH databases, the **admin\_info** control file is automatically kept consistent within itself, and with the other GASH files. Renaming a user who is also a GASH administrator will make the appropriate changes in the **admin\_info** file. Changes to the **hosts\_info** database can effect changes in the **admin\_info** e-mail addresses.

### mail\_control

The **mail\_control** file contains a list of event codes that correspond to significant events that can occur during the execution of the GASH control program. Upon such an event, GASH consults the **mail\_control** file to construct a list of e-mail addresses to which notification should be sent.

### networks\_by\_room

The **networks\_by\_room** control file contains a mapping of room numbers to IP networks. It is used by the GASH systems commands to present a choice of valid IP networks when a system is created or modified, or to verify a proper IP address when a system is moved from room to room. This file must contain a record for a room before a system can be placed in that room by GASH.

### pending\_actions

The **pending\_actions** control file contains a record of actions to be taken by GASH at some point in the future, one per line. Each pending action entry contains a time stamp for the action, a type field indicating the kind of action to be performed, an optional field indicating the object that the action is to be performed on, and a list of users to send mail to when the action is performed.

Events in the **pending\_actions** file may lead to modifications in one or more GASH databases, or may cause mail to be sent to users or administrators reminding them of an action to occur in the future. Typically, any database modification event will be preceded by a set of mail events, one per week until the time of the event.

The **pending\_actions** file is processed every night during automated GASH processing. Any events scheduled to occur are performed and removed from the **pending\_actions** file. If an event could not be performed, the record for the action is left in the **pending\_actions** file and mail is sent to the system administrators overseeing GASH.

### shell\_paths

The **shell\_paths** control file contains a list of login shell choices to be presented to GASH administrators creating or modifying a user account. Administrators may enter arbitrary shells, but problems may occur with ftp if the shell chosen is not in a client's `/etc/shells` file. **internet\_assignment**

The **internet\_assignment** control file contains a list of system types recognized by GASH. This list will be presented to GASH administrators creating or modifying a system. Each system type is associated with an IP subrange corresponding to a portion of an IP class C network or class B subnet. This allows, for instance, all routers to be assigned IP addresses with a host number between 254 and 255. The **internet\_assignment** file allows both ascending and descending ranges to be defined.

### Limitations of the GASH Control Program

While GASH is extremely functional, it does have several more or less arbitrary limitations and restrictions as currently implemented. GASH will allow users and systems to be renamed, but it will not allow groups, netgroups, or volumes to be renamed. GASH has some support for augmenting automounter home maps with the NIS '+' syntax, but has no support within the editor for removing or re-ordering such augmentation lines. With the exception of the **admin\_info** control file, all GASH control files must be set up and maintained by hand.

While it has not yet become a significant problem at ARL:UT, the single-user limitation could prove to be a significant bottleneck in very large installations. Various avenues to making GASH multi-user have been discussed, and in the future we may produce some kind of multi-user GASH.

GASH does not yet have any way of creating users' home directories on NCE volumes. Instead, when a user is added to a home volume, moved from home volume to home volume, or renamed, GASH will send mail to the administrators of the systems involved, describing the actions that need to be performed. Because remote human administrators are still involved at this point, some GASH actions do not have immediate turn-around.

There are many areas of GASH that would benefit from a richer user interface. The GASH commands accept command line parameters, but this facility is not very well developed. The list commands in particular would be well served by command line switches to specify search criteria. Many places in GASH will attempt to prompt the user with a reasonable list of alternatives at a prompt, but most do not. The linear quality of the current GASH interface makes most tasks very straightforward, but a graphical user interface would have significant advantages in a number of these areas. It would be convenient for GASH to allow an administrator to browse the GASH databases in a separate window or session while working with an editing command. It would also be nice to arrange for some kind of scripting language interface for GASH.

### GASH NCE Administrative Restrictions

The GASH control program enforces a number of administrative restrictions that are designed to make the divisions of control and responsibility within the NCE immediately apparent. GASH requires that administrator uid and gid ranges be contiguous. This restriction makes transferring users or groups between administrative groups non-trivial, but makes it feasible to allow a group of GASH administrators to share responsibility for a set of users and groups. Group and netgroup names must start with a three letter administration code that can be used to immediately identify the department or

group to which the group or netgroup belongs. GASH currently requires that group names be six characters in length.

GASH insists on assigning all uid's, gid's, and IP addresses. This guarantees that the id's and addresses are allocated in such a way as to insure that a previously used id or IP address will not be re-used until all other free id's or IP addresses in the appropriate range have been used. Unfortunately, this means that the initial construction of the GASH databases must largely be done by hand. Fortunately, the database consistency checks in GASH are extremely helpful in establishing a comprehensive and consistent NIS and DNS regime. In general, the lack of automated support for transitioning into the GASH environment is the weakest area of the GASH implementation as it currently stands.

#### Observations on implementing the GASH NCE at ARL:UT

Before we developed and implemented the GASH NCE, computing resources at ARL:UT were poorly integrated. Several groups within the laboratory had independently developed NIS and DNS domains, and record keeping and account administration were fragmented throughout the laboratory. NFS file sharing was possible only within local groups, and software management effort was being duplicated several times over. Network records were difficult to keep up to date, and local network configuration changes were often not promptly reported to the central network administrators.

Within the groups, the complexity of maintaining the many administrative databases by hand was becoming unmanageable. Even in the ARL:UT Computer Center, user and group records were poorly synchronized between computers.

These problems do not exist today. The GASH NCE has provided significant, concrete advantages for all involved. A common NIS and NFS regime allows much easier access to data across the network than was previously available. The GASH program makes complex system administration tasks trivial, and giving this power to local group administrators makes turn-around time for administrative tasks much faster. Our records are complete and up to date, and remain so with minimal effort. Users can rely on having a common working environment on UNIX machines throughout the laboratory.

The transition to GASH was long and somewhat painful. We had to shuffle users and groups around, moving uid's and gid's into organized blocks. A great deal of data entry and data verification had to be performed. Our experience with the great mass of data entry required led directly to the automated cross-database consistency checks that are now part of GASH. In addition to data entry and conversion, the effort of developing

and documenting GASH itself was not trivial. The gains in organizational clarity have been exceptional, and remain so even as the number of users and networked machines continues to increase rapidly.

#### What About NIS+?

NIS+ [7,8] is a drastically expanded and enhanced product compared to NIS. It provides significantly better security, and more intelligent database update propagation. NIS+ contains a mechanism for splitting NIS domains into hierarchical subdomains, a mechanism aimed at solving the same data-space partitioning problem that GASH was developed to address. We began developing GASH before NIS+ was announced, but it turns out that the mechanisms that NIS+ provides in this area do not adequately address our needs.

The main capability that GASH provides over standard NIS+ is the ability to split control over an NIS domain by uid, gid, and object name. NIS+ only allows domain splitting via subdomains, with different client systems belonging to different subdomains. It is not possible with the standard NIS+ mechanisms to distribute administrative control over a single machine. At ARL:UT, we have a SPARCcenter 2000 that is used by researchers from several different research groups. With GASH, the research groups are allowed to control their own users, groups, and e-mail aliases. With standard NIS+, such control could only be afforded to researchers with their own group of workstations.

Until such time as NIS+ client software is available on all common UNIX platforms, ARL:UT will need to support an NCE that supports standard NIS clients. This means that even when we go into NIS+ with GASH, we will not be able to make use of the hierarchical domain system provided by NIS+. Migration to NIS+ for the enhanced security and intelligent database propagation features is inevitable. NIS+'s hierarchical domains and intelligent update propagation may provide a useful substrate for a multi-user GASH (with one administrator per subdomain concurrently), and might make supporting multiple DNS domains more straightforward.

#### Anticipated Future Developments

We plan on enhancing our NIS support for e-mail by supporting a `mail.generics` NIS map, that will map NCE user names to their outgoing e-mail signature. We are also ultimately planning to make the overhaul of GASH required to integrate GASH with an NIS+ central server.

We are interested in developing GASH into a more fully automated tool, with RPC protocols for automating remote volume administration. In the more long term, we can see a significant benefit from reworking the basic architecture of GASH around an object oriented database system. A good

deal of GASH's code is devoted to maintaining cross-database consistencies that could be handled automatically by an object oriented database system. A graphical user interface or batch / script interface would be significantly easier to implement with such a base architecture.

We currently run code during GASH's nightly processing which generates the CSO e-mail directory for the laboratory. We are interested in doing more work within the ARL:UT environment to connect the GASH databases to our Personnel Office's databases, so that user accounts can be validated against Personnel Office records.

#### Availability

Preliminary user documentation is available for World Wide Web access from URL [http://www.arlut.utexas.edu/csd/gash\\_docs/gash.html](http://www.arlut.utexas.edu/csd/gash_docs/gash.html). The GASH software itself is available through this web server. The `opt_depot` software is available for World Wide Web access from URL [http://www.arlut.utexas.edu/csd/opt\\_depot/](http://www.arlut.utexas.edu/csd/opt_depot/).

#### Credits

GASH is the product of long effort on the part of many people at ARL:UT, both within the Computer Science Division and within the GASH user community. GASH could not have happened without the support of the systems administration and user communities within ARL:UT.

The GASH NCE was primarily conceived and designed by Dan Scott. The initial design of the GASH control program was developed by Dan Scott and Dean Kennedy. Dean Kennedy was the original implementor of the GASH control program, and is responsible for much of the program's infrastructure and all of its complex systems support. The author designed and developed the e-mail alias support, implemented large portions of the final code, and acted as a clearing house for bugs and Things That Needed To Be Done. Pug Bainter is the mastermind behind the GASH - NIS interface code and the finer points of GASH's netgroup and system support. Pug is also responsible for having nailed down exactly what an NCE client needs to do to play ball with GASH. Kristi Hennard implemented the initial pending\_actions processor and helped determine how GASH should behave so that job accounting would work out properly. Navin Manohar worked diligently to create WWW documentation for GASH. Alison Louise Brown and Jim Phillips worked long and hard to show us many new and interesting bugs that we were not aware of, and to manage the transition of our user and group databases into the brave new world of GASH. Virgil Moore, Pat Flinn, Pug Bainter, Joe Collins and several group administrators did the same for some 980+ network devices that are running around the laboratory. Dan Scott, Frank Douma, Jay Scott, and Elaine McSwain helped see

GASH through over two years of development with advice, supervision, and patience.

#### Author Information

Jonathan Abbey is a system analyst in the Computer Science Division of Applied Research Laboratories, The University of Texas at Austin, where he has worked since September 1989. He graduated from The University of Texas at Austin with a B.S. in computer science in May 1993. His e-mail address is [jonabbey@arlut.utexas.edu](mailto:jonabbey@arlut.utexas.edu) (a GASH generated alias!). General queries regarding the GASH software or the GASH NCE should be directed to [gash-authors@arlut.utexas.edu](mailto:gash-authors@arlut.utexas.edu).

#### References

- [1] Stern, H. "Managing NFS and NIS", O'Reilly & Associates, Inc., 1991.
- [2] Albitz, P., Liu, C. "DNS and BIND", O'Reilly & Associates, Inc., 1993.
- [3] Callaghan, B., Lyon, T. "The Automounter", Sun Microsystems, Inc. White Paper.
- [4] Allman, E. "SENDMAIL Installation and Operation Guide Version 8.35 (For Sendmail Version 8.6)".
- [5] Wong, W., Colyer, W. "Depot: A Tool for Managing Software Environments", *Proc. LISA VI*, October, 1992.
- [6] Allman, E. "SENDMAIL Installation and Operation Guide Version 5.1.3++ (For Sendmail Version 5.67a+IDA-2.0)".
- [7] "NIS to NIS+ Transition", Sun Microsystems, Inc., December 1993.
- [8] "SunOS 5.3 Administering NIS+ and DNS", Sun Microsystems, Inc., October 1993.



## THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- fostering innovation and communicating both research and technological developments,
- providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its annual technical conference, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter ;*login*;, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with the MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in ;*login*;.

### SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association, devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well. There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided.

USENIX Association membership services include:

- Subscription to ;*login*;, a bi-monthly newsletter;
- Subscription to *Computing Systems*, a refereed technical quarterly;
- Discounts on various UNIX and technical publications available for purchase;
- Discounts on registration fees to the annual technical conference and tutorial programs and to the periodic single-topic symposia;
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- The right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association:

ANDATACO  
ASANTÉ Technologies, Inc.  
Frame Technology Corporation  
Matsushita Electrical Industrial Co., Ltd.  
Network Computing Devices, Inc.

OTA Limited Partnership  
Quality Micro Systems, Inc.  
Tandem Computers, Inc.  
UNIX System Laboratories, Inc.  
UUNET Technologies, Inc.

*SAGE Supporting Member:*  
Enterprise Systems Management Corporation

For further information about membership, conferences or publications, contact:

The USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA

Email: [office@usenix.org](mailto:office@usenix.org)  
Phone: +1-510-528-8649  
Fax: +1-510-548-5738

